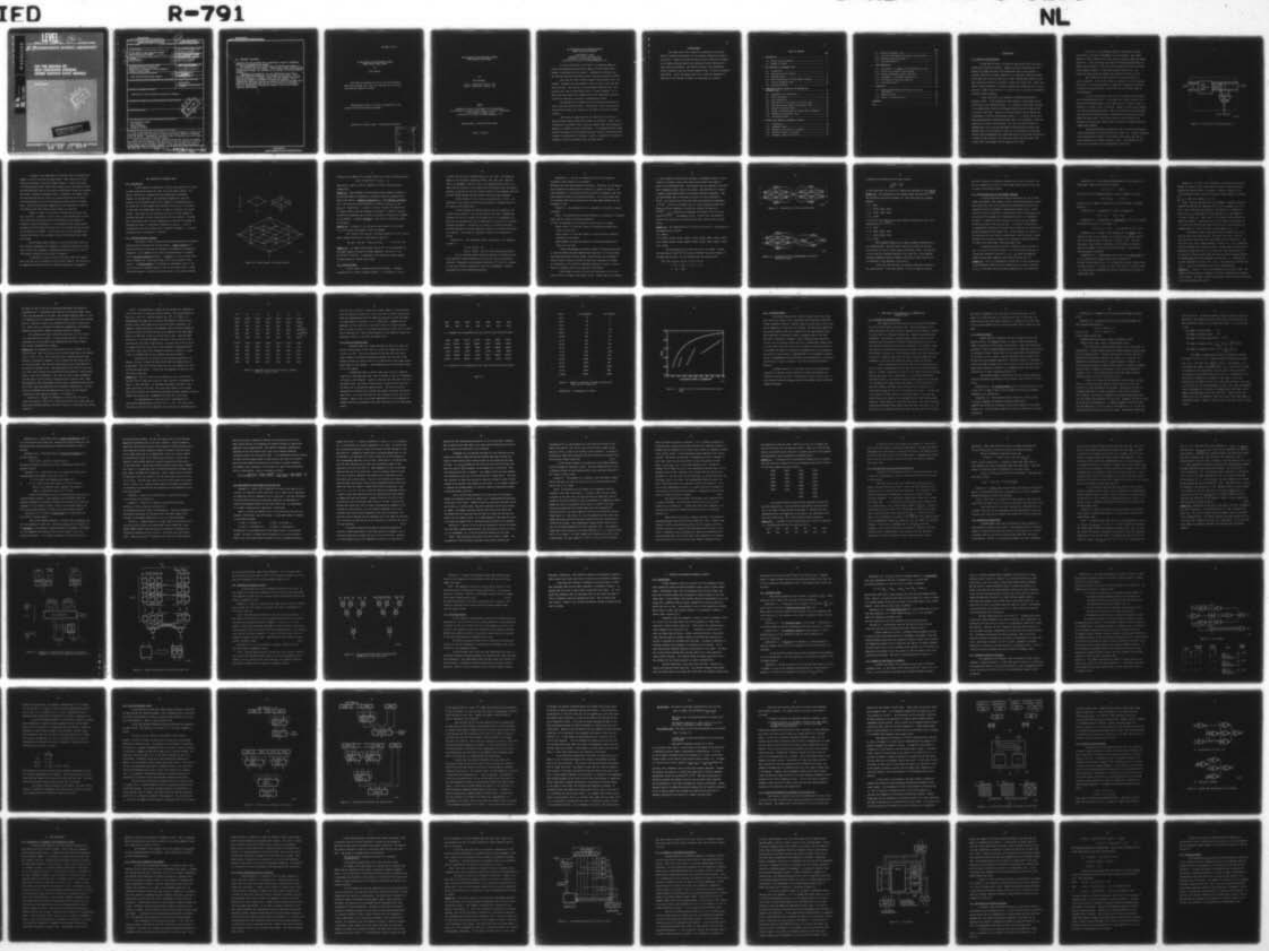
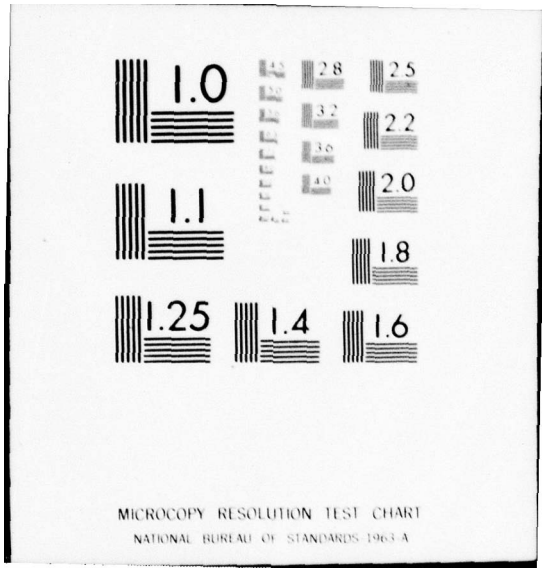
AD-A056 280 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
ON THE DESIGN OF SELF-CHECKING SYSTEMS UNDER VARIOUS FAULT MODE--ETC(U)
OCT 77 J DUSSAULT DAAB07-72-C-0259
UNCLASSIFIED R-791 NL

1 OF 2
AD
AD56 280




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

LEVEL II

REPORT R-791 OCTOBER, 1977 UILU-ENG 77-2238

CSL COORDINATED SCIENCE LABORATORY

ON THE DESIGN OF SELF-CHECKING SYSTEMS UNDER VARIOUS FAULT MODELS

JEAN DUSSAULT

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

ON THE DESIGN OF SELF-CHECKING SYSTEMS UNDER VARIOUS FAULT MODELS.

5. TYPE OF REPORT & PERIOD COVERED

Technical Report

6. PERFORMING ORG. REPORT NUMBER

R-791, UILU-ENG-77-2238

7. AUTHOR(s)

Jean Dussault

8. CONTRACT OR GRANT NUMBER(s)

DAAB 07-72-C-0259

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

12 | 16p

11. CONTROLLING OFFICE NAME AND ADDRESS

Joint Services Electronics Program

12. REPORT DATE

Octe[illegible]77

13. NUMBER OF PAGES

97

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

15. SECURITY CLASS. (of this report)

UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Fault-Tolerant Design
Self Checking
Coding Techniques
Arithmetic Circuits

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

One of the objectives of this work is to study the design of totally self-checking systems that are made up of blocks whose fault behavior is different from one another. Consequently, different codes are mentioned and their associated circuit structures discussed in detail.

Codes that are used to protect against unidirectional errors are studied. Systematic and non-systematic codes are shown to have the same basic structure. The structure of non-systematic unordered codes, more precisely the class of fixed-weight codes, if further examined. It is shown that these codes have

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

20. ABSTRACT (continued)

codewords that can be effectively classified in terms of congruence, cycling and complementation classes.

The capabilities of unordered codes are defined with respect to circuit structure and fault models. Some results, based on the previous classification scheme, are presented on the design of two-level minimal checkers.

Some practical suggestions for and limitations on the use of unordered codes under different fault models are described. After a brief introduction to arithmetic codes, previous concepts about self-checking are adapted to arithmetic circuits and checkers. It is demonstrated that known checkers and adders are TSC provided some basic rules are followed and some increase in the hardcore is tolerable. Finally a discussion of translators attempts to unify the different codes and fault models.

UILU-ENG 77-2238

ON THE DESIGN OF SELF-CHECKING SYSTEMS UNDER VARIOUS FAULT MODELS

by

Jean Dussault

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259.

Approved for public release. Distribution unlimited.

ON THE DESIGN OF SELF-CHECKING SYSTEMS UNDER VARIOUS FAULT MODELS

BY

JEAN DUSSAULT

B.Sc.A., Université d'Ottawa, 1973
M.A.Sc., Université d'Ottawa, 1974

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1977

Thesis Adviser: Professor Gernot Metze

Urbana, Illinois

ON THE DESIGN OF SELF-CHECKING SYSTEMS UNDER VARIOUS FAULT MODELS

Jean Dussault, Ph.D.
Coordinated Science Laboratory and
Department of Electrical Engineering
University of Illinois at Urbana-Champaign, 1977

One of the objectives of this work is to study the design of totally self-checking systems that are made up of blocks whose fault behavior is different from one another. Consequently, different codes are mentioned and their associated circuit structures discussed in detail.

Codes that are used to protect against unidirectional errors are studied. Systematic and non-systematic codes are shown to have the same basic structure. The structure of non-systematic unordered codes, more precisely the class of fixed-weight codes, is further examined. It is shown that these codes have codewords that can be effectively classified in terms of congruence, cycling and complementation classes.

The capabilities of unordered codes are defined with respect to circuit structure and fault models. Some results, based on the previous classification scheme, are presented on the design of two-level minimal checkers.

Some practical suggestions for and limitations on the use of unordered codes under different fault models are described. After a brief introduction to arithmetic codes, previous concepts about self-checking are adapted to arithmetic circuits and checkers. It is demonstrated that known checkers and adders are TSC provided some basic rules are followed and some increase in the hardcore is tolerable. Finally a discussion of translators attempts to unify the different codes and fault models.

ACKNOWLEDGMENT

The author would like to express his gratitude to his thesis advisor, Professor Gernot Metze, for his guidance during the research of this thesis. Further, the author would also like to thank his colleagues of the Digital Systems Group at the Coordinated Science Laboratory for their help and friendship. The technical support of the Coordinated Science Laboratory, and the clerical support of Mrs. H. Corray are greatly appreciated. Finally the author would like to thank the Department of Education of the Province of Québec for supporting this research.

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Survey of the Literature

Around 1956 the study of reliability was pursued within the scope of automata and information theories. Among the results established then, Von Neumann [53], Shannon and Moore [49] demonstrated that it was possible to construct arbitrarily reliable computers out of unreliable computing elements. In the same vein, Elias has shown that a simple combinational computer can be made arbitrarily reliable by encoding its inputs and outputs [18]. These three papers form the foundation of reliable computation including the area of self-checking: indeed, error detection is a special case of reliability improvement employing coding techniques.

By the time results were available the IBM 650 was already in existence (1955). It was perhaps the first machine to contain what are now called totally self-checking elements: its one-digit decimal adder consisted of an inverter-free circuit with biquinary-coded inputs and outputs as well as two-rail carry [48]. However all validity checkers did not satisfy the requirements of totally self-checking checkers, i.e., a faulty checker could have passed erroneous information without proper error signalling. Hence the system was not totally self-checking. This type of overall structure is characteristic of all the work performed in the 1955-1968 period and some hereafter and will be referred to simply as self-checking or self-testing. Sellers, Hsiao and Bearnson give a good description of the theory and practices followed in that time span. The codes most widely used were parity [48], residue [33], fixed weight [19] and algebraic [41] codes.

As the area of self checking advanced, some progress was made concurrently in the area of redundancy i.e. fault masking. Most common examples of these techniques are triple modular redundancy (TMR) and quadded logic [39]. In the late sixties the STAR (Self-Testing and Repairing) [5] computer drew attention to both areas as they merged to produce the dynamically redundant computer. In the early seventies Carter et al. also studied the logic design for dynamic and interactive recovery [12,13].

After 1968 self-testing continues to be used extensively, at least in practice. As for as the theory is concerned it is still discussed [56,57] but it is often presented as a compromise or imperfect alternative to totally self-checking which, at that time, had just formally made its first appearance.

A totally self-checking (TSC) circuit is characterized informally by the following design rules: 1) the normal set of input regularly tests the circuit including checkers for any modeled fault, 2) when an output is produced it is either correct or non-code. Figure 1.1 gives a block diagram of a totally self-checking circuit. Most of the work has been concentrated upon such circuits characterized by an inverter-free structure and a fixed weight code space input. This type of circuit is well suited for the detection of unidirectional faults which are intrinsic to memories and some communication channels.

Carter and Schneider introduced the idea of totally self-checking circuits and applied it to parity and two-rail codes [11]. Anderson formalized it by introducing the notions of self testing, fault-secure and code disjoint properties later refined by Smith [1,2,52]. The remainder of this section is an overview of the work generated in this area.

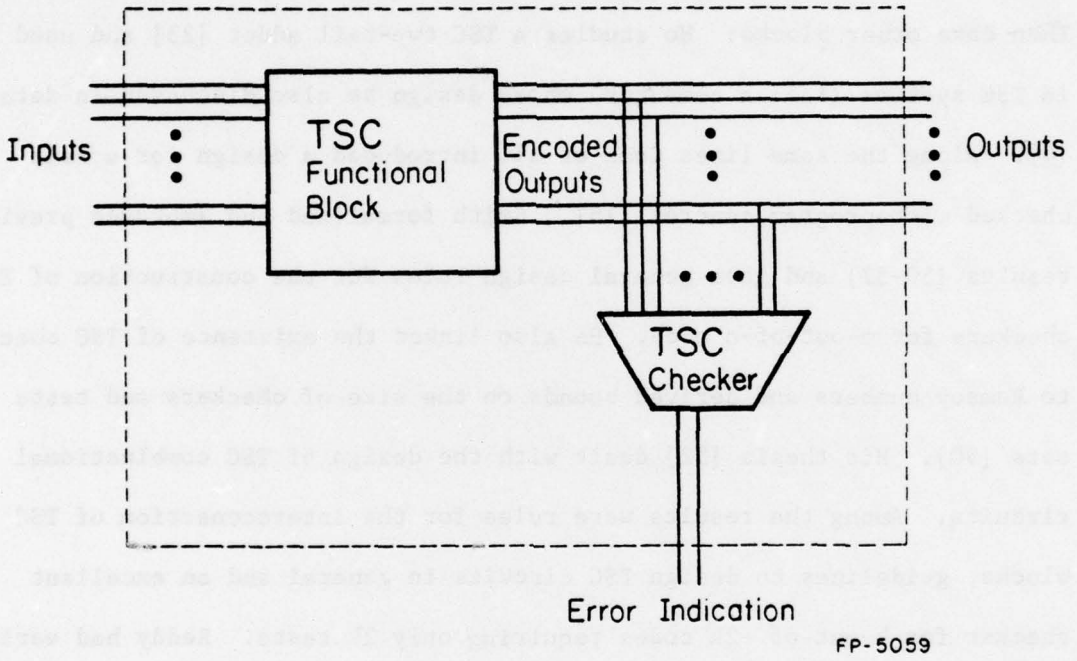


Figure 1.1. The totally self-checking model [1].

Among the first results generated in that area were designs of TSC parity checkers and duplication comparators (Carter and Schneider [11], Anderson and Metze [2]). These were simply exclusive-or trees and were TSC with respect to the occurrence of single faults [2,11]. At the same time designs of two-rail checkers were also presented, TSC this time with respect to unidirectional faults [1,11]. Anderson then provided a general method to construct checkers for m-out-of-n codes using majority functions [1,2]. Then came other blocks: Ho studies a TSC two-rail adder [23] and used it in TSC systems (i.e. a computer) whose design he also discussed in detail [4]. Along the same lines Cook et al. introduced a design for a self-checked microprogram control [15]. Smith formalized and improved previous results [50-52] and gave general design rules for the construction of TSC checkers for m-out-of-n code. He also linked the existence of TSC checkers to Ramsey numbers and derived bounds on the size of checkers and tests sets [50]. His thesis [52] dealt with the design of TSC combinational circuits. Among the results were rules for the interconnection of TSC blocks, guidelines to design TSC circuits in general and an excellent checker for k-out-of -2k codes requiring only 2k tests. Reddy had earlier presented more restricted results, namely TSC checkers for k-out-of-2k, $(k+1)$ and k-out-of-$(2k+1)$ codes [44] and an easily testable realization for p and (p or more)-out-of-n codes [44].

Besides fixed weight codes there is a class of systematic unordered codes (Berger codes and their extensions [8]) which has received some attention recently. The advantage of these codes is that the information digits are separate from the check digits. Reddy [46] and Smith [52] have both proposed schemes to construct TSC checkers for these codes but so far the results obtained are not as satisfactory as the results

accumulated in the case of m-out-of-n codes. The fact is that separability does not enhance the ease of construction of TSC circuits for these codes.

As far as the design of TSC sequential machines using coding techniques is concerned, the tendency is toward asynchronism and is exemplified in the works of Diaz [17], Pitt [40], Ozgüner [33], and Reddy and Kuhl [29]. An alternative to coding techniques to achieve TSC design is the use of time redundancy schemes (i.e. alternating logic) proposed and studied by Bark and Kinne [7], Reynolds [47], and Woodard [58]. Synchronous sequential machines have also been studied in that context [47,58].

1.2. Summary of Thesis

One of the objectives of this thesis is to study the design of totally self-checking systems that are made up of blocks whose fault behavior (and therefore also the input and output codes used) is different from one another. Consequently, different codes will be mentioned and their associated circuit structures discussed in detail.

In the next chapter, codes that are used to protect against unidirectional errors are presented. Systematic and non-systematic codes are shown to have the same basic structure. Indeed they can be produced using the same generation rules that are expressed in terms of products of lattices. The structure of non-systematic unordered codes, more precisely the class of fixed-weight codes, will further be examined. It is shown that these codes have codewords that can be effectively classified in terms of congruence, cycling and complementation classes. These results will be used in the following chapters.

In Chapter 3 the capabilities of unordered codes are defined with respect to circuit structure and fault models. Some results will be presented on the design of two-level minimal checkers. In fact the problem of finding partitions that checkers must satisfy will be discussed in detail and some guidelines based on the previous classification scheme are presented. The concept of growth and existence tests is extended to the design of TSC circuits and using this as a basis it is shown that there do not exist 2-level TSC checkers for a class of systematic unordered codes, the Berger codes. Finally some practical suggestions for and limitations on the use of unordered codes under different fault models are described.

Chapter 4 touches an area where the design of TSC circuits has been restricted: arithmetic circuits. After a brief introduction to arithmetic codes, previous concepts about self-checking will be adapted to arithmetic circuits and checkers. It is demonstrated that known checkers and adders are TSC provided some basic rules are followed and some increase in the hardcore is tolerable. At the same time it will be apparent that practical TSC checkers are not known to exist if the hardcore size cannot be increased.

The following chapter attempts to unify the different codes and fault models by presenting some thoughts on translators and ways of using codes to their best advantage. Encoding and decoding circuits as well as a TSC memory architecture will be given in examples.

In conclusion, a brief look at the pin fault model will explain why it was kept out of contention in the previous chapters. The thesis will be summarized and finally topics for further research will be suggested.

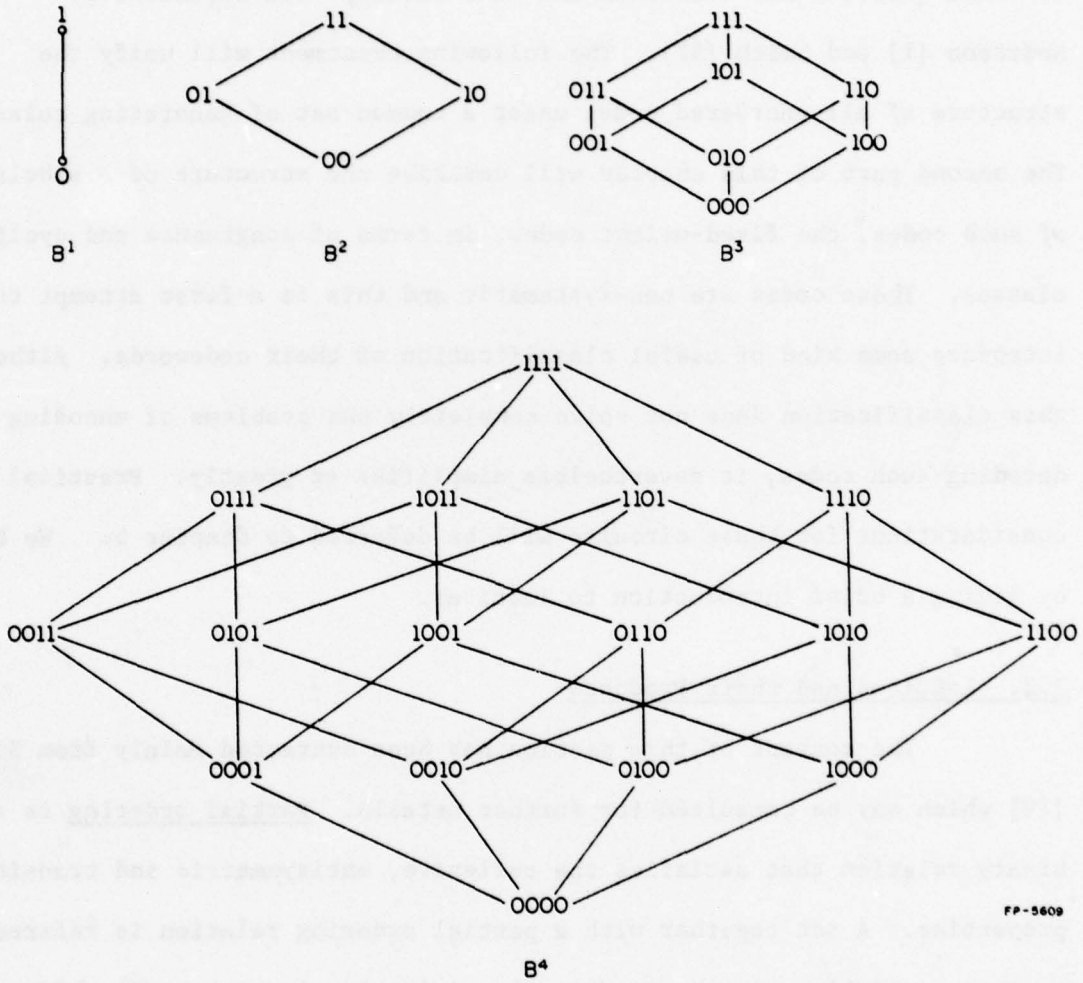
2. THE STRUCTURE OF UNORDERED CODES

2.1. Introduction

In this chapter we examine the structure and properties of a class of codes using lattice theory and other tools from modern algebra. The codes in question are unordered and have already been discussed by Anderson [1] and Smith [52]. The following treatment will unify the structure of all unordered codes under a common set of generating rules. The second part of this chapter will describe the structure of a subclass of such codes, the fixed-weight codes, in terms of congruence and cycling classes. These codes are non-systematic and this is a first attempt to introduce some kind of useful classification of their codewords. Although this classification does not solve completely the problems of encoding and decoding such codes, it nevertheless simplifies it greatly. Practical considerations for those circuits will be deferred to Chapter 5. We begin by giving a brief introduction to lattices.

2.2. Lattices and their Products

The content of this section has been extracted mainly from Birkhoff [10] which may be consulted for further details. Partial ordering is a binary relation that satisfies the reflexive, antisymmetric and transitive properties. A set together with a partial ordering relation is referred to as a partially ordered set (poset). A lattice is a poset in which every pair of elements has a least upper bound (l.u.b.) and a greatest lower (g.l.b.). A Boolean algebra is a distributive complemented lattice. The set of vertices of a Boolean lattice is the set of binary vectors. Binary vectors with exactly k ones will be called k-tuples. The partial ordering



B^1 B^2 B^3 B^4

Figure 2.1. Hasse diagrams of Boolean lattices.

relation on the elements of a Boolean lattice is the usual covering relation:

$$\underline{a} \leq \underline{b} \text{ if and only if } a_i \leq b_i \quad \forall i$$

where the a_i's and b_i's are the components of binary vectors $\underline{a}$ and $\underline{b}$, respectively.

Example 2.1: Hasse diagrams of some Boolean lattices appear in Figure 2.1. We say that vertex $\underline{a}$ is adjacent to vertex $\underline{b}$ if they are connected by an edge in the lattice. Immediate predecessors $\underline{a}_i$'s and immediate successors $\underline{c}_i$'s to vertex $\underline{b}$ are vertices adjacent to vertex $\underline{b}$; also $\underline{a}_i \leq \underline{b}$ and $\underline{b} \leq \underline{c}_i$ $\forall i$ respectively. A subset of vertices is said to be ordered and is called a chain if any two elements are adjacent. A subset of vertices is said to be unordered and is called an antichain if every pair of elements is not adjacent. A maximal antichain is such that its cardinality cannot be increased.

Example 2.2: In Figure 2.1 all antichains consisting of all k-vertices (i.e. all vertices with exactly k ones) are maximal.

Let A and B be two lattices. The direct product A × B is a lattice whose elements form the set of ordered pairs $(\underline{a},\underline{b})$ and

$$(\underline{a}_i, \underline{b}_i) \circ (\underline{a}_j, \underline{b}_j) = (\underline{a}_i \circ \underline{a}_j, \underline{b}_i \circ \underline{b}_j) \qquad \circ = \text{g.l.b or l.u.b}$$

Example 2.3: All higher order Boolean lattices B^i, $i > 1$ are isomorphic to the direct product of lattices B^j whose exponents j add up to i: e.g. $B^4 = B^2 \times B^2 = B^3 \times B^1$ etc. A subproduct is any subset of a direct product and may neither be a lattice nor a chain.

2.3. Unordered Codes

We first equate unordered codes with antichains. A maximal unordered code is simply a maximal antichain. It is possible to look at

a lattice and pick out an antichain and use it as a code. For example in Figure 2.1 the antichain consisting of 0011, 0101, 1001, 0110, 1010 and 1100 is the (maximal) 2-out-of-4 code. This approach however gives no insight into the structure or the construction of unordered codes from smaller lattices, and it does not tell how closely related all the unordered codes are. We shall instead combine smaller lattices using very simple mappings.

By the converse of l.u.b. (g.l.b.) we mean g.l.b. (l.u.b.). By inspection it can easily be seen that the converse of a Boolean lattice is the lattice itself upside down.

In the following only Boolean lattices will be considered. Let A and B be such lattices. If B is the converse of A then a complementary antichain of A in B is located "at the same level" as its complement in A. This idea of complementarity will be generalized to the notion of relatively unordered antichains. Intuitively it is easier to visualize relatively unordered antichains as being "comparably located" in a lattice with respect to a converse. The concept of comparable location can be formalized as follow:

Definition 2.1: Two antichains α and β, respectively, are comparably located if:

$$\forall \alpha_i \geq \alpha \quad \exists \beta_i \geq \beta \quad \text{and}$$

$$\forall \alpha_i \leq \alpha \quad \exists \beta_i \leq \beta \quad \text{and vice versa with } \alpha_i \in A \text{ and } \beta_i \in B.$$

If A is a Boolean lattice and B the converse of a Boolean lattice then comparably located antichains in A and B are said to be relatively unordered. Comparably located maximal antichains in B^k and in its converse are not only relatively unordered but are also complementary. Relative unordering can also be defined functionally:

Definition 2.2: Two sets of elements are said to be relatively unordered if their product is an antichain.

Unordered codes have been defined as antichains. This point of view implies that antichains are selected from a lattice or a product of lattices. Another approach is to use smaller lattices and define a subproduct that maps them simply and exactly into desired antichains. It is easily shown that all unordered codes can be constructed in the same manner whether they are systematic or not.

Theorem 2.1: Any unordered code is a subproduct of relatively unorderd antichains.

Proof: It follows directly from the definition of antichains (= unordered codes) and the concept of relative unordering.

In fact, the mappings can be performed in several ways, for example:

-Every element in A and every element in relatively unordered antichains in B are used.

-Every element in A and some elements in every relatively unordered antichain in B are used.

-Some elements in A and some elements in relatively unordered antichains in B are used.

Clearly the mappings are listed in order of decreasing efficiency or rate since fewer and fewer elements are being used. As it can be seen the construction rules are very broad. Since lattices can be defined in terms of smaller lattices it turns out that their antichains can always be constructed out of relatively unordered antichains of smaller lattices. Hence all unordered codes can be obtained in that manner.

Fixed weight or m-out-of-n codes are codes made up of all the binary vectors of length n with exactly m ones. Berger codes are systematic,

i.e. they comprise all the vectors of length n (information vector) to each of which is attached a check vector whose value reflects the number of zeroes in the information bits. Fixed weight codes are produced by the first type of mappings, Berger codes by the second. More precisely let $\times$ denote a complete concatenation product of relatively unordered antichains. Then all fixed weight codes of the form k-out-of-(k + j) can be obtained as $B^k \times B^j$. As a special case k-out-of-2k codes are produced most efficiently as $B^k \times B^k$. In the case of Berger codes it involves mapping complete antichains of B^k with singletons of relatively unordered antichains of $B^{\lceil \log_2 k \rceil}$. Let $*$ represent such mappings; then Berger codes are of the form $B^k * B^{\lceil \log_2 k \rceil}$. Two examples follow. In both cases, concatenation products are indicated by the arrows and the small letters label the antichains.

Example 2.4: The construction of a 3-out-of-6 code from B^3. Using Figure 2.2 the mappings are as follow:

$a \times \delta$: 000111

$b \times \gamma$: 001011, 001101, 001110, 010011, 010101, 010110, 100011, 100101, 100110

$c \times \beta$: 011001, 011010, 011100, 101001, 101010, 101100, 11001, 110010, 110100

$d \times \alpha$: 111000

The construction of the 3-out-of-6 code is not unique. However the construction from $B^3 * B^3$ is the most "efficient" one since lattices of the same size are used. All the codewords are thus produced since

$$|B^3 * B^3| = |a \times \delta| + |b \times \gamma| + |c \times \beta| + |d \times \alpha|$$
$$= 1 \times 1 + 3 \times 3 + 3 \times 3 + 1 \times 1$$
$$= 20 = \binom{6}{3}.$$

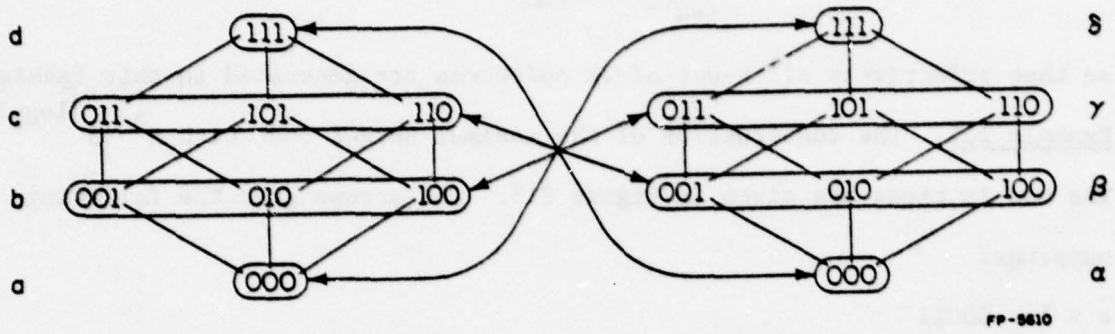


Figure 2.2. Construction of the 3-out-of-6 code.

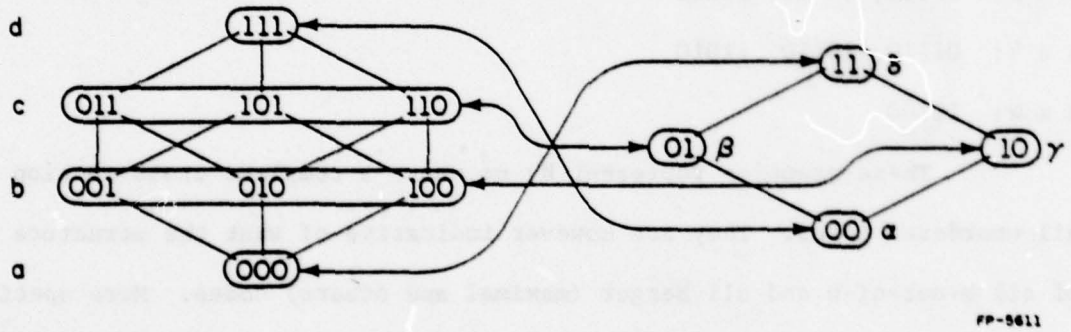


Figure 2.3. Construction of the maximal Berger code with 3 information digits.

In general this satisfies the well-known identity

$$\sum_{i=0}^{k} \binom{k}{i}^2 = \binom{2k}{k}$$

so that effectively all k-out-of-2k codewords are generated in this fashion.

Example 2.5: The construction of the maximal Berger code with $B^3 * B^{\lceil \log_2 3 \rceil}$. The two lattices are given in Figure 2.3. The arrows give the following mappings:

a × δ: 00011

b × γ: 00110, 01010, 10010

c × β: 01101, 10101, 11001

d × α: 11100

It can also be seen that the only other maximal unordered code than can be obtained from $B^3 * B^{\lceil \log_2 3 \rceil}$ is:

a × δ: 00011

b × β: 00101, 01001, 10001

c × γ: 01110, 10110, 11010

d × α: 11100

These examples represent by no means a complete cross-section of all unordered codes. They are however indicative of what the structure of all m-out-of-n and all Berger (maximal and others) codes. More specifically we have not considered mappings of the third type. These mappings are most inefficient (they are subsets of the first 2 types) and will not be considered in the remainder of this thesis.

It has just been seen that all unordered codes have essentially the same structure. In the next chapter, it will be shown that despite

this fact, checkers for different unordered codes do not look alike. But first we examine the structure of fixed-weight codes which are widely used in totally self-checking systems.

2.4. The Classification of Fixed-Weight Codewords

In recent years there has been an increase in the use of fixed-weight codes mainly in the design of totally self-checking digital circuits. Even though these codes are nonordered and nonsystematic they were also considered earlier for asymmetric communication channels [19]. In this section we present a method for classifying some of the fixed weight codes into classes thus bringing the encoding method closer to a "systematic" procedure. An approach to the construction of such codes was suggested by Kautz and Elpsas a few years ago [25]. It involved solving problems in combinatorics related to the Steiner Triple system and its generalization, balanced incomplete block designs [22]. As they pointed out, results in that area were not very useful in coding theory. Here we present a method by which one can produce a complete set of generators without repetition for some of the fixed weight codes with higher rates.

As mentioned earlier m-out-of-n codewords contain precisely m ones in a total of n digit positions. Let the positions of the ones in a particular codeword be labelled $c_1, c_2, \ldots, c_m$, where the numbering scheme of these digit locations starts with 0 and goes up to n-1.

Example 2.6: The c_i's corresponding to 011010 are 1,2 and 4 respectively. In the following, codewords will be designated by an m-tuple consisting of the c_i's obtained from their binary representation, e.g. 011010 → 124

Definition 2.2: Let us define the permutation T on the elements of a fixed-weight codeword (m-out-of-n) as follows:

$$T(c_i) = c_i + 1 \qquad \text{mod } n$$

This notation can be extended naturally to sets of elements as follow:

$$T(C) \equiv T(c_i) \qquad \forall\, c_i \in C$$

the set of c_i's to which T is applied is called the ensemble of elements moved by T.

Definition 2.3: A permutation T is said to be cyclic if

$$T(c_i) = c_i + 1 \qquad \text{mod } n \quad \forall i \leq m$$

Definition 2.4: A permutation T is quasi-cyclic if the set of elements left fixed by the cycle T is non-empty i.e.

$$T(c_i) = c_i + 1 \mod n \qquad \text{for some (but not all) } i \leq m.$$

Definition 2.5: A (quasi) cyclic permutation T is ordered if the elements of the cycle are ordered. In the case of non-negative integers this is the same as cycles of the form $\begin{pmatrix} 0 & 1 & 2 & 3 & \ldots & m \\ 1 & 2 & 3 & 4 & \ldots & 0 \end{pmatrix}$. This last definition is made only for convenience as there exists an isomorpism between the elements of cycles with identical length. Using concepts just mentioned we can define:

Definition 2.6: A fixed-weight code is said to be (quasi-)cyclic if it is closed under an ordered (quasi-)cyclic permutation T.

We shall see that they all are. Let us denote the greatest common divisor of m and n by (m,n). If (m,n) = 1, m and n are said to be relatively prime. It is now possible to establish some results concerning the structure of fixed weight codes.

Theorem 2.2: If $(m,n) = 1$ the codewords in an m-out-of-n code can be grouped into exactly $\binom{n}{m}/n$ classes each of which is closed under cyclic shifting and contains exactly n codewords. Moreover a complete set of $\binom{n}{m}/n$ generators can be listed by finding the m-tuples whose c_i's satisfy $\sum_{i=1}^{m} c_i \equiv j \bmod n$ for a fixed j, $0 \leq j < n$.

Proof: First we know there are $\binom{n}{m}$ codewords. Suppose we pick any codeword and find that the summation of its c_i's is congruent, say, to b mod n. Cyclic shifting T corresponds to the addition of 1 (modulo n) to each of the c_i's or equivalently to the addition of m to the summation ($\equiv$ b mod n) of the c_i's. Since $(m,n) = 1$, the length of the cycle will be exactly the modulus n. Hence there are $\binom{n}{m}/n$ classes of cardinality n. It is easy to see from the above that each class will contain exactly one codeword with $\Sigma c_i \equiv j \bmod n$ for each j in the range 0 to 2k. □

This theorem covers, among others, codes of higher rates of the form $m \pm 1$-out-of-2m, m-out-of-$2m \pm 1$, those with least rate, 1-out-of-m (m-1-out-of-m), those of the form "even"("odd")-out-of-"odd"("even"). The fact that we are using a fixed congruence relation, i.e. j mod n guarantees that the solutions are in distinct classes. Clearly there are other sets of generators; indeed there are $n\binom{n}{m}/n$ such sets. Note that when half of the m-out-of-n codes have been found the other half can be obtained directly by complementing each of the codewords in every code, i.e. replacing them by their complement in the set $\{0,1\ldots,m-1\}$. This procedure is obvious since $\binom{n}{m} = \binom{n}{n-m}$ and corresponds to bit by bit complementation.

Example 2.7: Table 2.1 gives all the codewords of a 3-out-of-7 codes. The elements of each column are congruent to the value indicated by the heading of the columns. The order in which they are given follows the natural ordering of the cycle staring with 0 mod 7.

T^0	T^1	T^2	T^3	T^4	T^5	T^6
0 mod 7	3 mod 7	6 mod 7	2 mod 7	5 mod 7	1 mod 7	4 mod 7
016	012	123	234	345	456	056
034	145	256	036	014	125	236
025	136	024	135	246	035	146
124	235	346	045	156	026	013
356	046	015	126	023	134	245

Table 2.1 - The codewords in a 3-out-of-7 code.

Theorem 2.3: If $(m,n) \neq 1$ then the m-out-of-n code can be constructed as follow:

1) Construct the m-out-of-(n-1) code using Theorem 2.2.

2) If $((m-1),(n-1)) \neq 1$ then set $m = m-1$, $n = n-1$ and reapply 1.

3) If $((m-1),(n-1)) = 1$ then

   3a) construct the (m-1)-out-of-(n-1) code using Theorem 2.2.

   3b) concatenate the current n-1 to the current (m-1)-out-of-(n-1) code

   3c) merge the resulting code with the current m-out-of-(n-1) code.

4) Set $m = m+1$, $n = n+1$ and reapply 3b until the originally looked for m-out-of-n code is obtained.

In other words the code can be constructed recursively and when recursion is exhausted we have a classification for the codewords of the m-out-of-n code. The code thus obtained will be quasi-cyclic.

Proof: If $(m,n) \neq 1$ then $(m,n=1) = 1$. It follows that the m-out-of-(n-1) code can be constructed using the previous theorem. The subcode thus produced will consist of all m-vertices which do not have n-1 as a digit position. (Remember that the range of digit positions is 0 to n-1.) The next step consists of producing all the (m-1)-out-of-(n-1) vertices and concatenate them with the digit (n-1). The merging of these two subcodes produces exactly all the codewords of the m-out-of-n code. Numerically this is verified by the well-known recursion formula for binomial coefficients:

$$\binom{n-1}{m} + \binom{n-1}{m-1} = \binom{n}{m}$$

Of course m-1 and n-1 may not be relatively prime hence the reason for reapplying step 1. Eventually these terms will be relatively prime since m and n are decremented all the time. The ultimate situation where m = 1 and n is anything ((1,n) = 1) guarantees a finite number of steps in the recursive process. Typically, however, the recursion will not have to proceed that far. Steps 3 and 4 are provided to guide the concatenation and subsequent merging in cases where a number of recursion levels is required.

The procedure given above constructs recursively the (m-1)-out-of-(n-1) and m-out-of-(n-1) codes. At each recursion interval a new bit is concatenated so that ultimately the codewords have the desired length. This is best illustrated by an example.

Example 2.8: The construction or classification of the codewords for the 3-out-of-9 code. The 3-out-of-8 and the 2-out-of-8 codes have to be realized $(\binom{8}{3} + \binom{8}{2} = \binom{9}{3})$. The 3-out-of-8 code is constructed first using Theorem 2.2 yielding 56 codewords. This is going to be the cyclic part of the 3-out-of-9 code. The 2-out-of-8 code cannot be built using Theorem 2.2 so the work is divided in two: the realization of the 1-out-of-7 and 2-out-of-7 codes, both of which can be constructed using Theorem 2.2 (this process yields a total of 28 codewords). The 1-out-of-7 code is then concatenated with the digit 7 and the result is merged with the 2-out-of-7 code. These 28 codewords are then concatenated with the digit 8 (forming the quasi-cyclic part of the code) and merged with the 56 codewords obtained above. All the 3-out-of-9 codewords have been obtained since none of the 56 codewords constructed at the beginning contained a 1 in position 8.

Corollary 2.4: The set of codewords in a k-out-of-2k code can be sorted into exactly $2\binom{2k-1}{k-1}/2k-1$ classes, half of which are closed under the quasi-cyclic shifting operation and half of which are closed under the cycling operation T.

Proof: (by construction) Suppose we have obtained the codewords of a (k-1)-out of-(2k-1) using the previous theorem. (We can use Theorem 2.2 since (k-1, 2k-1) = 1). We first observe that $\binom{2k}{k} = 2\binom{2k-1}{k-1}$. By concatenating the number 2k-1 to each of the codewords of the (k-1)-out-of-(2k-1) code, exactly half of the k-out-of-2k code is obtained. All congruence properties mod 2k-1 are preserved, that is,all the elements in this half of the k-out-of-2k code are still congruent to the same number mod 2k-1 as they were in the (k-1)-out-of-(2k-1) code. Since the digit position 2k-1 is constant throughout this half of the k-out-of-2k code, this portion of the code is quasi-cyclic. The other half of the k-out-of-2k is made up of the k-out-of-(2k-1) code obtained by complementing the first half; since none of the codewords in this half contains a c_i equal to 2k-1 we conclude that all the codewords have been obtained □.

This corollary is perhaps the most interesting application of Theorem 2.3 since it shows how simply fixed codes with highest rate can be generated. They are such that exactly half of the codewords are cyclic and half are quasi-cyclic. In fact all the codewords are cyclic in the first (2k-1) positions.

Example 2.9: Table 2.2 lists all the codewords in the 4-out-of-8 code. The first half is simply the 3-out-of-7 code to which is concatenated the digit 7. Since the digit 7 is stationary throughout, this part is the quasi-cyclic part of the code. The cyclic part consists of the second half and its elements are obtained by satisfying the same congruence relation. Simpler yet,one just has to complement the first half of the code.

By complementation is meant bit by bit complementation or equivalently complementation in the set $\{0,1,\ldots,n-1\}$. For codes of the form k-out-of-2k half of the codewords can be obtained by complementing the

T^0	T^1	T^2	T^3	T^4	T^5	T^6	
0 mod 7	3 mod 7	6 mod 7	2 mod 7	5 mod 7	1 mod 7	4 mod 7	
0167	0127	1237	2347	3457	4567	0567	From Table 2.1
0347	3457	2567	0367	0147	1257	2367	
0257	1367	0247	1357	2467	0357	1467	
1247	2357	3467	0457	1567	0267	0137	
3567	0467	0157	1267	0237	1347	2451	
0 mod 7	4 mod 7	1 mod 7	5 mod 7	2 mod 7	6 mod 7	3 mod 7	
2345	3456	0456	0156	0126	0123	1234	
1256	0236	0134	1245	2356	0346	0145	
1346	0245	1356	0246	0135	1246	0235	
0356	0146	0125	1236	0234	1345	2456	
0124	1235	2346	0345	1456	0256	0126	

Table 2.2 Codewords of the 4-out-of-8 codes as obtained from the 3-out-of-7 code.

the first half as long as it makes up an integer number of cycling classes. If we now define the classes of a k-out-of-2k code in terms of quasi-cyclic shift and complementation then the number of generators is half the number of classes given by Corollary 2.4 i.e. $\binom{2k-1}{k-1}/2k-1$. Complete sets of generators for the 4-out-of-9, 5-out-of-10, 5-out-of-11 and 6-out-of-12 codes are given in Table 2.3 (the digit should add up in this case to 0 mod 9 and 11 respectively). Finally some statistics on the number of classes are presented in Table 2.4 for codes with higher rates.

2.5. Rates of Unordered Codes

Unordered codes have enough redundancy to detect all single and multiple unidirectional errors (that is, errors that change only 0s to 1s or only 1s to 0s). This can be explained by the fact that such errors cause elements of relatively unordered antichains (that are used in the product realizing the code) to move in the same direction and become ordered with respect to some codeword. Code capabilities will be discussed further in the next chapter.

Although they detect a fairly large class of errors, unordered codes have a reasonably good rate. Rate is defined as $\log_2$ (# of codewords)/(# of bits in the cordword). Graph 2.4 plots the rates for both k-out-of 2k and Berger codes. The graph is defined only at discrete points and curves are drawn only to indicate the general aspect. The price to pay for separability is evident from the graph since Berger codes have consistently lower rates. It is also seen that the rate increases with the size of the codeword. However, this is more than compensated for by the complexity of hardware that has to manipulate those codes (especially for the fixed weight codes).

0126	0135	0234	0378	0468	0567	1278
1368	1458	1467	2358	2367	2457	3456

a - A complete set of generators for the 4-out-of-9 and 5-out-of 10 codes

01235	012910	013810	014710	01485	015610	01579
01678	023710	02389	024610	02479	02569	02578
034510	03469	03478	03568	04567	068910	123610
12379	124510	12469	12478	12568	13459	13468
13567	158910	167910	23458	23467	248910	257910
267810	347910	356910	357810	36783	45789	456810

b - A complete set of generators for the 5-out-of-11 and 6-out-of-12 codes.

Table 2.3.

(m,n)	# of codewords	# of classes
(2,5)	10	2
(3,6)	20	2*
(3,7)	35	5
(4,8)	70	5*
(4,9)	126	14
(5,10)	252	14*
(5,11)	462	42
(6,12)	924	42*
(6,13)	1716	132
(7,14)	3432	132*
(7,15)	6435	429
(8,16)	12870	429*
(8,17)	24310	1430
(9,18)	48620	1430*
(9,19)	92378	4862
(10,20)	184756	4862*

Table 2.4: Number of codewords and number of classes for some m-out-of-n codes (m,n).

*Permutation - complementation classes.

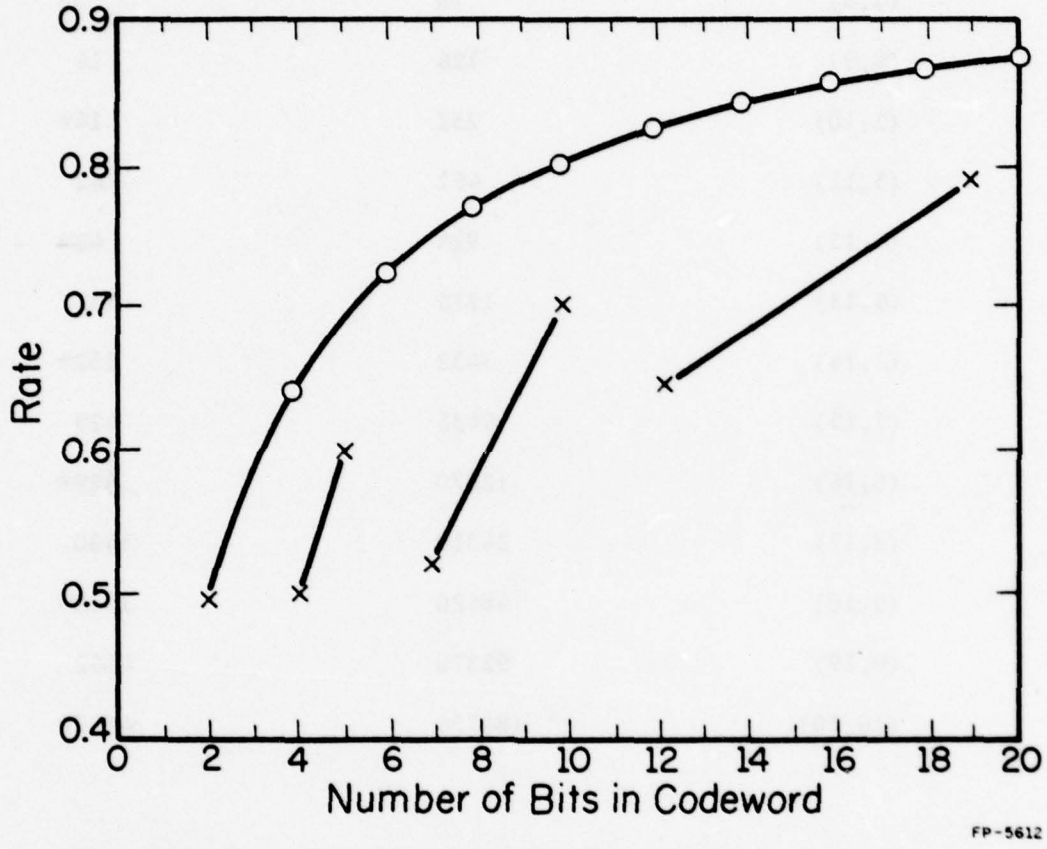


Figure 2.4. Relative rates for best fixed-weight and Berger codes.

2.6. Concluding Remarks

The previous theorems and corollary grouped fixed weight codes into classes and complete sets of generators were obtained by satisfying a fixed congruence relation. Nothing, however, has been said as far as the actual computation of these sets is concerned. The exhaustive computation can be quite tedious (even for a computer) but once the generators have been found there is obviously no need to recompute them all the time. (The proof of Theorem 2.2 suggests a way of obtaining those generators.) In fact a way of encoding a binary word into a fixed weight codeword would be to use some of the bits of that word to address a table and other bits to determine the amount of shifting and complementation to be performed on the contect of that location. Complementation is an inexpensive operation so that in implementation both permutation and complementation, if possible, should be used as basic constructs. This will be discussed further in Chapter 5.

Although codewords in a particular code can be systematically classified, no general relation between the classes and generators of different codes has been found. This would be a useful relation since it would simplify functional mappings between fixed weight codes of different lengths and weights.

3. SOME RESULTS ON THE DESIGN OF TSC CHECKERS FOR UNORDERED CODES

3.1. Introduction and Definitions

This chapter presents some results on the design of TSC checkers for both the fixed-weight and Berger codes. Two-level checkers are in general desirable since they can detect malfunctions most rapidly. However for codes with only a moderate codeword size, fanout restrictions become prohibitive. The problem of finding minimal two-level checkers can be identified with the problem of finding a minimal partition on the set of code vertices and this partition must be satisfied by all checkers. There are difficulties associated with checkers for Berger codes and they will be discussed in Section 3.6. The following sections give some consideration to the use of unordered codes under the single fault and asymmetric error assumptions. But first some definitions about circuits are provided.

In the following we shall present some results pertaining to the class of acyclic combinational circuits that totally self-checking checkers make up. Circuits, denoted by the letter G, will perform functional mappings from an input space X to an output space Y. In a network with several functional blocks G_i, the notion of input and output spaces can be extended so it emcompases all network inputs and outputs. For circuits with r inputs and s outputs, the domain will be the set X of 2^r vertices of the r-cube and the range the set Y of 2^s vertices of the s-cube. In order that it be possible to detect failures, only proper subsets of the input and output spaces may be applied to, and produced by the circuit during normal operation. These subsets are called the code spaces. The circuit then performs partial mappings from a domain called the input code space A ($A \subset \{2^r$ vertices$\}$) onto a codomain called the output code space B($B \subset \{2^s$ vertices$\}$).

The relative complement of the code spaces A and B with respect to the spaces of 2^r input vertices or 2^s output vertices will be referred to as input and output non-code spaces, respectively. Elements of the code spaces will be called codewords and all remaining elements will be called non-codewords.

3.2. Errors and Faults

Perhaps the most widespread concept of distance in use is the Hamming distance. The Hamming distance is simply a metric measuring the number of positions in which binary vectors differ. Codes with capabilities specified in terms of minimum Hamming distance also assume that error patterns are completely random. However there are situations where the distribution of the faults is not so random. Memories and busses tend to exhibit error patterns that have dependent components. Similarly, failures of integrated circuits may affect multiple lines and are likely to be in the same direction. These types of faults are referred to as asymmetric faults and can be best characterized in terms of polarized distance [1]:

The polarized weight is defined only on error vectors obtained by borrowless arithmetic subtraction of the erroneous vector from the "correct" vector.

Definition 3.1: The polarized weight $\omega_p(E)$, of an error vector E is the 2-tuple (ω^+, ω^-) where ω^+ and ω^- are the numbers of positive and negative components in E, respectively.

Definition 3.2: The polarized distance between two vectors $\underline{x}$ and $\underline{y}$ is $\omega_p(\underline{x}-\underline{y})$ (component by component borrowless subtraction).

Anderson has carried out an extensive study of the properties of polarized distance [1]. For the present purposes it suffices to provide the following definitions:

Definition 3.3: Asymmetric errors have polarized weights such that $\omega^+ \neq \omega^-$.

Definition 3.4: Unidirectional errors have polarized weights with the following characteristics:

$$\text{if } \omega^+ \neq 0 \text{ then } \omega^- = 0$$

and vice versa $\quad \text{if } \omega^- \neq 0 \text{ then } \omega^+ = 0.$

One may readily observe that

-Unidirectional errors form a subset of asymmetric errors.

-Single errors form a subset of unidirectional errors.

Up to this point we have mentioned errors only in the context of coding theory. In logic networks these will appear as hardware failures which are of a permanent nature. Transient failures may also occur and will be detected if the propitious conditions are present i.e. if and only if a codeword that sensitizes the fault is applied during its lifetime. So, for all practical purposes, transient faults either will appear as permanent stuck-at faults or simply be non-existent, as far as fault modeling is concerned. This is the classical stuck-at fault model where lines get solidly stuck-at-1 (s-a-1) or stuck-at-0 (s-a-0).

A fault in a circuit will be denoted by the set $\{\ell_i/d_i,\ 1 \leq i \leq k\}$ where ℓ_i is usually some gate input or output line stuck at the value $d_i \in \{0,1\}$; k is the multiplicity of the fault. The fault set $\mathfrak{F}$ is the set of modeled faults and is usually justified in terms of both simplicity and likelyhood. The notions of asymmetric and unidirectional errors carry over very simply to the definition of fault sets. Asymmetric and unidirectional errors may have any multiplicity. In addition asymmetric faults are such that the number of $d_i = 0$ is not equal to the number of $d_i = 1$; in the case of unidirectional faults all the d_i's are equal. Single faults simply have

unit multiplicity. The absolute number of faults of different types are given below. The qualifier "absolute" means that circuit structure and fault equivalence classes are not taken into account. In a circuit with n lines:

The number of single faults = $2n$

The number of multiple faults = 3^n-1

The number of unidirectional faults = $2(2^n-1)$

The number of symmetric faults = $\sum_{\text{even } i}^{n} \binom{n}{i}\binom{i}{i/2} = \sum_{i}^{\lfloor n/2 \rfloor} \binom{n}{i}\binom{n-i}{i}$

The number of asymmetric faults = $(3^n-1) - \sum_{\text{even } i}^{n} \binom{n}{i}\binom{i}{i/2}$

The number of unidirectional faults is obtained by computing the number of all 0's or all 1's patterns with the multiplicity ranging from 1 to n. This comes out to $2\sum_{i=1}^{n} \binom{n}{i} = 2(2^n-1)$. For symmetric faults one computes the number of symmetric faults ($\binom{i}{i/2}$ term) times the number of patterns with even multiplicity ($\binom{n}{i}$ term with i even) and performs the summation of the product terms for all even i's. Equivalently we can calculate the summation of the number of lines taken i at the time, multiplied by the number of lines stuck in the other direction, also taken i at the time, but out of the remaining n-i lines. The number of asymmetric faults is presented as a difference to emphasize that this number and the number of multiple faults are of the same order. Indeed circuits with complete test sets for asymmetric faults will cover a very large number of the multiple faults. Figure 3.1 gives an idea of how some classes of faults overlap.

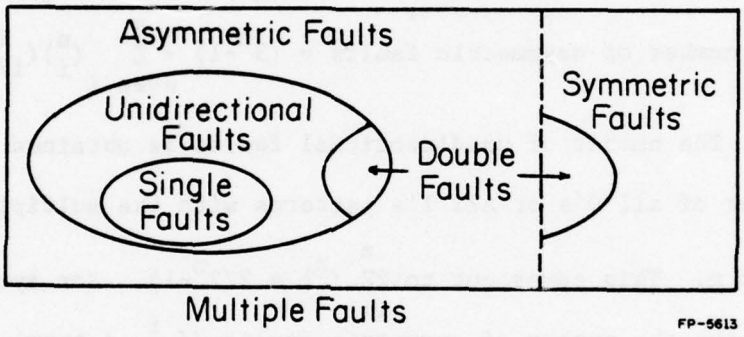


Figure 3.1. Classes of faults.

In a circuit with 4 lines, about 22.5% of all multiple faults will be symmetric whereas for a circuit with 20 lines the figure is about 11% and less than 8% for a circuit with 40 lines. This means that, ignoring structural constraints, a circuit with about 40 lines, designed to be totally self-checking with respect to asymmetric faults, will be self-checking with respect to more than 92% of all possible faults. From a coding theory standpoint, if information is encoded (using unordered codes) in blocks of 40 bits, then on the average the code will detect 92% of all errors. Unfortunately the code will not correct any error.

3.3. Basic Definitions

In this section, basic concepts and results relevant to the scope of this chapter as well as the rest of the thesis are presented. The formalism in the following is due to Smith [49-51].

As before let A be the input code space and B, the output code space. The functional block G will perform a mapping $G(a,f)$ where the first argument is some member of the input space and f is an element of the fault set $\mathfrak{F}$. Mappings under the no fault situation are denoted by $G(a,\phi)$. The following definitions are made with respect to a fault set $\mathfrak{F}$.

Definition 3.5: A functional block is <u>fault secure</u> if

$$\forall f \in \mathfrak{F} \quad \forall a \in A \quad G(a,f) = G(a,\phi) \text{ or } G(a,f) \notin B$$

In words: G will never produce, under a modeled fault, an incorrect codeword.

Definition 3.6: A functional block is <u>self-testing</u> if

$$\forall f \in \mathfrak{F} \quad \exists a \in A \quad \text{such that} \quad G(a,f) \notin B$$

That is, all modeled failures can be detected by some codeword.

Definition 3.7: A functional block is totally self-checking (TSC) if it is fault secure and self-testing. Removing the overlap between the self-testing and fault secure properties an equivalent definition of TSC is obtained.

Definition 3.8: A functional block is totally self-checking if it is fault secure and

$$\forall f \in \mathfrak{F} \ \exists a \in A \quad \text{such that} \quad G(a,f) \neq G(a,\phi)$$

An equally interesting concept in the idea of essentially totally self-checking circuits.

Definition 3.9: A functional block G is essentially totally self-checking (ETSC) with respect to the fault set $\mathfrak{F}$ if:

i) it is totally self-checking with respect to $\mathfrak{F}$.

or ii) it is fault secure with respect to $\mathfrak{F}$ and for any $f \in \mathfrak{F}$ which is not tested by input codewords, the network G under f is ETSC with respect to $\mathfrak{F}^* = \{\mathfrak{F}-f\}$.

Totally self-checking circuits for unidirectional faults are constructed without inverters. They implement unate functions; these are completely monotonic and therefore preserve the direction of the fault and propagate it to the outputs [10,49]. Totally self-checking checkers are TSC circuits whose function is defined by the code disjoint property:

Definition 3.10: A circuit is code disjoint if it always maps non-code inputs to noncode outputs.

This is the original definition. Actually some non-codewords may never be produced by a fault set. These non-codewords will be referred to as improbable. The property of mapping probable non-codewords (and perhaps not all improbable non-codewords) into non-code outputs is the error preserving property. It is equally satisfying in practical checkers as

the code disjoint property. One may also require that error preserving mappings be performed under the no fault condition. However asymmetric faults may afflict both a circuit and its checker, in which case the fault-secure and code disjoint properties combine to produce a non-code output. In the single fault case it is assumed that a fault in a circuit implies the exclusion of a fault in its checker and vice versa. In that case a checker is code disjoint under the no-fault situation and fault-secure in the faulty mode. Totally self-checking checkers usually have 2 outputs; this figure is a minimum and there is nothing conceptually incorrect with more outputs (although less economical). These 2 outputs will be produced by functions F and G, be labelled (f,g) and take normal values in the set $\{(0,1),(1,0)\}$. Non-code space inputs or faults in the checker will produce $(f,g) \in \{(0,0),(1,1)\}$. Smith has shown that the following two conditions are necessary and sufficient for the realization of TSC checkers for m-out-of-n codes [50,52].

Condition 1: Every $(m+1)$-vertex covers at least one m-vertex in F and at least one m-vertex in G.

Condition 2: Every $(m-1)$-vertex is covered by at least one m-vertex in F and at least one m-vertex in G.

The problem has now the flavour of combinatorics. Indeed the existence of satisfactory partitions for arbitrary m and n has been linked to the existence of the Ramsey number $N(m+1, m+1; m) = R(m)$ as follows:

Theorem 3.1: (Smith [50-52]) for $m \leq (\frac{n}{2})$, $R(m) > m$ if and only if the m-vertices of the n-cube can be partitioned to meet conditions 1 and 2.

Ramsey numbers known in this case are $R(1) = 3$ and $R(2) = 6$ inferring the existence of checkers for the 1-out-of-2, 2-out-of-4, and 2(3)-out-of-5 codes. Since $13 \leq R(3) \leq 17$, checkers for the 3-out-of-n codes certainly

exist for $6 \leq n \leq 12$, perhaps for $13 \leq m \leq 17$ and definitely not for $n > 17$. These results did not tell anything new; however they give an idea of the magnitude of the design problem. The problem of finding a satisfactory partition of the code vertices into F and G is in general difficult (relatively easier for k-out-of-2k codes). If a satisfactory partition exists, let H(m,n) be the minimal number of code vertices that must be in block F. Then the 2-level AND-OR realization of F has H(m,n) + 1 gates; similarly the OR-AND 2-level realization of G has H(m,n) + 1 gates (such realizations will be called hybrid). H(m,n) is bounded as follow:

$$H(m,n) \geq \max\left(\left\lceil \frac{n}{m} \left\lceil \frac{n-1}{} \left\lceil \cdots \left\lceil \frac{n-m+3}{3} \left\lceil \frac{n-m+2}{2} \right\rceil \right\rceil \right\rceil \right\rceil, \left\lceil \frac{n}{n-m} \left\lceil \frac{n-1}{n-m+1} \left\lceil \cdots \left\lceil \frac{m+3}{3} \left\lceil \frac{m+2}{2} \right\rceil \right\rceil \right\rceil \right\rceil \cdots \right\rceil\right)$$

3.4. Some Results on the Design of Two-Level TSCC

Theorem 3.2: Given a set of implicants of size m such that all (m-1)-vertices are contained in these implicants, let us form a set of implicates by taking the relative complement of these implicants. Then the irredundant two-level hybrid realization using these implicants and implicates is a totally self-checking checker for the m-out-of-2m code. By irredundancy it is meant no duplication of implicants or implicates.

Proof: As usual for checkers it has to be proven that they are code disjoint, fault-secure, and self-testing. As mentioned before it suffices to prove the following:

1) f(m) = 1 iff g(m) = 0 2) f(m) = 0 iff g(m) = 1

3) f(m-1) = 0 and g(m-1) = 0 4) f(m+1) = 1 and g(m+1) = 1

The function f is the AND-OR part of the checker while g is the OR-AND section. By f(m) it is meant that the domain of f is a set m-vertices. Cases 1 and 2 are easily proven using the relative complement constraint.

Suppose first f(m) = 1. Clearly an implicant is equal to 1, i.e m variables are 1, and therefore its relative complement is all zeros. It follows that the corresponding implicate is zero thus forcing the final AND gate to zero, hence g(m) = 0. Inasmuch the second case is concerned suppose that at least one variable in an implicate is 1, then at least one of the variables in the corresponding implicant is set to zero. Since g(m) = 1, it implies that each and every implicate has at least one on-variable causing all the implicants to be zero thus f(m) = 0. These arguments are dual in the other direction and need not be given here. Case 3 is equally simple to prove. If an (m-1) vertex is applied to the checker then no implicant will be true since they all have size m. Also all (m-1)-vertices are subsets of some implicants and because of the relative complement property each (m-1)-vertex will have no element in common with at least one implicate. Thus g(m-1) = 0. For case 4 one has to show that there is an implicant which is a subset of each (m+1)-vertex and that all implicates have some elements in common with each (m+1)-vertex. Suppose we have the set of all (m-1)-vertices and a set of m-vertices that covers all of them. Since we are dealing with closed sets then by simple complementation arguments we can say that all (m+1)-vertices (i.e. the relative complement of all (m-1)-vertices) covers the relative complement of the set of m-vertices. But both the set of m-vertices and its complement are of the same size and cover all m-1 vertices so that the set of all m-vertices is covered by the set of all (m+1)-vertices. Q.E.D.

The previous theorem establishes an equivalence between the problem of constructing hybrid 2-level TSCC and that of finding a set of m-vertices which has for subset the set of all (m-1)-vertices. Moreover it can be observed that if the cardinality of this set of implicants is

minimal then the corresponding realization of the 2-level TSSC is minimal. This situation occurs when there is no large overlap between the m-vertices. This is going to be formalized in the following

A minimal set of m-vertices such that all (m-1)-vertices are contained in it without repetition has the property that each m-vertex contains or covers exactly m(m-1)-vertices that are not covered by any other m-vertex. It can be verified for the 2-out-of-4 and 4-out-of-8 codes that there are exactly $\binom{2k}{k-1}/k$ members in the minimal set of m-vertices. However nothing can be said in general. For example m does not divide exactly the cardinality of the set of (m-1)-vertices for m larger than 5. Note that this situation is taken into account by using the ceiling of terms in H(m,n) instead of the ceiling of the product of the terms. It is, however, possible to approach the minimum number of m-vertices needed by reducing the amount of overlap between m-vertices.

Let us use the same notations as the one used in the previous chapter and represent vertices by numbers (c_i's) representing the positions of the ones in the vertex. Suppose we have two m-vertices such that all but one of their c_i's are the same, then both m-vertices have one (m-1)-vertex in common. For example vertices 12789 and 01278 both cover vertex 1278. Now assume that two m-vertices have the same c_i's except for two that are found to be not common to both. It is then possible to show that the set of (m-1)-vertices produced from or covered by those m-vertices is irredundant.

Theorem 3.3: The set of (m-1)-vertices produced by a set of m-vertices such that their elements,taken pairwise, are the same in all but two of the c_i's is irredundant i.e. no two (m-1)-vertices produced are the same.

Proof: Such m-vertices have exactly one (m-2)-vertex in common. The construction of (m-1)-vertices covering an (m-2)-vertex requires the

concatenation of a c_i which appears in no more than one m-vertex to the common (m-2)-vertex. Also the m-1 other (m-1)-vertices contain no more than one of the c_i's that do not belong to both m-vertices. It therefore follows that, under those conditions, the (m-1)-vertices are produced from the m-vertices without repetition. Q.E.D.

In the previous chapter it was shown how fixed weight codewords could be arranged in congruence classes. The next theorem provides an aid in finding a minimal or near minimal partition that satisfies conditions 1 and 2 based on those classes.

Theorem 3.4: The elements of a congruence class (excluding a complementary subclass) are such that any pair of such m-vertices have all but exactly two c_i's in common.

Proof: By definition all the elements in a congruence class are congruent to the same value modulo n. In order to move from one element to another in the same class, and thus preserve the congruence relation, changes to the digit positions c_i must be performed in a symmetrical fashion. Changes are said to be symmetric if the value c is added to one digit position and subtracted from another digit position to maintain the resulting vertex in the same congruence class (all the operations are performed modulo n). Some values of c will be inappropriate since they would change some digit positions into digit positions that are not going to be altered by the change. Such inappropriate changes produce vertices that shall be excluded automatically from the congruence class. The elemental change that can be effected is the simple addition of +1 to a digit position and of -1 to another. All other changes can be expressed in terms of that simple change. For this change of +1 and -1 it can easily be seen that

exactly two digit positions are affected. Also a systematic production of m-vertices from a single m-vertex that satisfies the congruence relation is possible and will generate all m-vertices that satisfy the congruence relation of the initial vertex. In this generation scheme complements will also be produced. The reason for excluding a complementary subclass in general is that 0 and n have the same value modulo n; hence the presence of both $0x_1x_2\ldots x_j$ and $x_1x_2\ldots x_jn$ in the same congruence class. Because its elements do not differ in two c_i's the complementary subclass that is to be excluded is one of the 2 subclasses that contains half the vertices just described i.e. for each vertex of the form $0x_1x_2\ldots x_j$ exclude the one of the form $x_1x_2\ldots x_jn$ or vice versa. The complementary subclass is empty for cyclic codes since 0 mod n is represented solely by the number 0. So, as long as that complementary subclass is excluded each pair of elements in the resulting set of m-vertices belonging to a congruence class will have exactly m-2 digit positions in common. Q.E.D.

The results of the previous two theorems can be expressed also in terms of distance. Two m-vertices that have all but two c_i's in common are at a distance of 4. Indeed since each vertex comprises a 1 in two positions that are not shared by the other vertex, the resulting distance is 4.

These results do not give a complete solution to the problem of generating minimal partitions that meet conditions 1 and 2. However they indicate not only a good initial guess but also the sequence in which a heuristic procedure to construct those partitions should proceed: First select a congruence class as starting set, remove the complementary subclass described above (if any) and select other codewords to be compared with the initial set in an order such that complete congruence classes

are examined one after the other. When this is over it is probable that some (m-1)-vertices will not have been covered. These (m-1)-vertices must then be covered with as few m-vertices as possible and this is the standard covering problem often encountered in switching theory.

Example 3.1: A starting set for constructing a minimal set of 5 vertices that will cover all 4-vertices can be the congruence class in the 5-out-of-10 code whose elements have value 0 mod 9:

01269	34578	12789	01278
01359	24678	13689	01368
02349	15678	14589	01458
03789	12456	14679	01467
04689	12357	23589	02358
05679	12348	23679	02367
		24579	02457
		34569	03456

The second and fourth columns are the complements of the first and third respectively. Also the third columns differs from the fourth only in the digit 9 being replaced by the digit 0. Therefore for each complementary pair one element has to be removed. The resulting set will cover $20 \times 5 = 100$ of the $\binom{10}{4} = 210$ 4-vertices without repetition.

Example 3.2: A minimal set of 4-vertices that will cover all 3-vertices i.e. a minimal partition for the 4-out-of-8 code is:

0123	0145	0167	0256	0247	0346	0357
4567	2367	2345	1347	1356	1257	1246

In this case most of the elements are congruent to 1 and 6 mod 7. This set has the property of being self-complementary. The two-level hybrid complementation of the checker is such that the inputs to the AND-OR section are exactly the same as those to the OR-AND section. Although this situation occurs also for the 2-out-of-4 code we are unable to conjecture anything about other codes.

3.5. Self-Checking and Growth and Existence Tests

In this section we extend the notion of growth and existence tests to the idea of self-checking. The next section will present an application of that approach.

These two types of tests were introduced by Paige [35] and by Kohavi [26] who called then "a" and "b' tests. We shall consider only the simplest form, i.e. tests for two-level structures. The "existence" tests verify the existence of an implicant. Growth tests consist in making implicants independent of a particular variable thus causing cubes to grow into larger ones. The no-fault result of the application of a growth test is an output equal to zero. More precisely suppose $f = \sum_{i=1}^{n} P_i$. In order to check that none of the inputs to the j^{th} AND gate is stuck at 0 one needs only to select an input vector $a_j \in P_j$ and $a_j \notin \sum_{i \neq j} P_i$. Also if the k-th input to the j^{th} AND gate is stuck at 1, the behaviour of the network changes as if a supplementary subcube P_{jk} (adjacent to P_j) was added to the original function. The test set is therefore the set of $b_{jk} \in P_{jk}$ such that $b_{jk} \notin \sum_i P_i$. Due to fault equivalence, the OR gate is completely tested by the above tests. Although there is one existence test per implicant, a growth test can check the growth of several implicants upon a single

application. These ideas extend very easily to design constraints for the self checking part of two-level totally self-checking circuits.

The set of existence tests must be such that

$$\forall y \ni f(y) = 1 \quad \exists a_j \in A \ni f(a_j) = f(y) \text{ and } a_j \notin \sum_{i \neq j} P_i$$

Dually f can be replaced by g. In words it simply means that each and every existence test must be performed using code space inputs. Also, the growth of an implicant must be checkable by a code space input:

Let $b_j = P_j$ and P_{jk} be adjacent boolean subcubes. Then these must exist input vectors

$$b_{jk} \in . \ni b_{jk} \subseteq P_{jk} \quad \forall k \text{ and } b_{jk} \notin \sum_{\forall i} P_i$$

Theorem 3.5: Assuming code disjoint design, the existence of "existence tests" is necessary and sufficient to guarantee that a 2-level irredundant AND-OR realization be totally self-checking.

Proof: Since $f \oplus g = 1$ if and only if their domain is some code space input then every growth test in f must be an existence test in g and vice versa

Since test sets for two level structures are sufficient (but not necessary) test sets for any irredundant multilevel realization of the same function, very little can be inferred from results for 2-level realizations.

3.6. Checkers for Berger Codes

Although Berger codes have the advantage of being separable, no good checkers are known for such codes. If the checkers have to be TSC with respect to unidirectional faults then the only method known is to translate the Berger code into an m-out-of-n code and then check that code using known checkers. One way is to translate the Berger code with n information

bits into the 1-out-of-2^n code and then check that code [46]. This is of course a very expensive approach for maximal codes with large n. Smith [51] maps the (3,5) Berger code onto the less redundant 2-out-of-4 codes. However no general methods are known to perform such mappings. If the fault model is the single fault assumption then the use of inverting gates is permitted and there is a variety of "simpler" ways to check Berger codes. One can use any irredundant structure that counts the number of ones in the information bits and compares it with the value of the check bits. Such a structure may be made up of half adders or be a unate array whose outputs produce the functions "m or more ones" combined with inverters to give a value that can be compared with the check bits [45].

It is in general recognized that ideal checkers should respond quickly to changes in their inputs. This is achieved by an implementation with the smallest number of levels. The next theorem is concerned with the non-existence of 2-level checkers for maximal Berger codes. What is less obvious is that it addresses essentially the same partition problem as for the fixed-weight codes.

Theorem 3.6: There do not exist 2-level realization of TSC checkers (with respect to unidirectional faults,i.e. inverter-free structure) for maximal Berger codes.

Proof: In maximal Berger codes there is a single codeword of the form $x_1 \ldots x_n x_{n+1} \ldots x_{n+\log_2(n+1)} = 0 \ldots 0 1 \ldots 1$. In order to check that codeword the checker under no fault must produce the output (f,g) = (0,1) or (1,0). Also because of unateness constraints the functions f and g are completely defined for all the non code inputs. For example the pair (f,g) for all inputs $x_1 \ldots x_n x_{n+1} \ldots x_{n+\log_2(n+1)} > 0 \ldots 0 1 \ldots 1$ is (1,1). Now suppose that

$f(0\ldots01\ldots1) = 0$ then there must exist codewords $y_1\ldots y_n y_{n+1}\ldots y_{n+\log_2(n+1)}$ such that $f(y_1\ldots y_{n+\log_2(n+1)})$ will cover the ones imposed by the code disjoint property. To preserve the unateness property and satisfy the requirement for an existence test, this will have to be achieved by a set of codewords whose check bits are at a distance 1 from the check bits of 0...01 ...1. This requirement is needed so that there are existence tests composed of single implicants that are also code vertices. The only such set is {{(n-1)-vertices}11...10}. The function f({(n-1)-vertices}11...10) must be 1 for it it is 0 it leaves uncovered a set of 1 in vertices of the form {(n-1)-vertices}111...1. So far so good. The next codewords to be covered by the function are {{(n-2)-vertices}11...101}. The check bits 11...01 are not adjacent ot the check bits 11...10. Therefore there is no way in which it is possible to cover by f the sets of 1s in the $\{(\underset{\text{vertices}}{n-2})\}$11...10 positions given that $f(\{(\underset{\text{vertices}}{n-1})\}11\ldots10)$ equals 0 and still have an existence test within the set of codewords. Dually f can be replaced by g and the theorem is proved in general. Q.E.D.

Example 3.3: Application of the proof to the Berger code with 7 information bits and 3 check bits. In the table below let a, b and c represent the check bits. The top row of digits gives the number of ones or the weight of the information bits. The squares represent code space outputs (0,1) or (1,0) and 0 and 1 stand for (0,0) and (1,1) respectively. All the non-code space inputs are completely defined by the unateness and code disjoint property.

abc	0	1	2	3	4	5	6	7
000	0	0	0	0	0	0	0	□
001	0	0	0	0	0	0	□	1
011	0	0	0	0	□	1	1	1
010	0	0	0	0	0	□	1	1
110	0	□	1	1	1	1	1	1
111	□	1	1	1	1	1	1	1
101	0	0	□	1	1	1	1	1
100	0	0	0	□	1	1	1	1

abc

Let f(0000000111) = 1. Then abc is the only unate function that can cover it. Its existence test is 0000000111 and the growth tests are subsets of {1-vertices} 110, {2-vertices} 101 and {4-vertices} 011. If we let f({1-vertices} 110) = 1 we do not have an essential growth test any more. Therefore let f({1-vertices} 110) be 0. The unate function g that will cover {1-vertices} 110 is ($\Sigma\Pi$(1 or more ones))·ab. The function f({2-vertices} 110) has to be covered by a unate function whose existence tests are ($\Sigma\Pi$(2 ones or more) ANDed with the consensus of 110 and 101. However such consensus does not exist thus precluding the existence of an existence test for the unate function that has to cover (2 or more ones)·(ab + ac) indicated by the dotted contour.

3.7. Design of Checkers under the Single Fault Assumption

Perhaps the most prevailing fault model being used is the single fault assumption. Totally self-checking systems have traditionally concentrated on other fault models as well. These other fault sets cover the single fault set as indicated in section 3.2, and it is the reason for the placement of this section. Smith presented a general treatment of the design

of circuits that use unordered code under the single fault assumption [52]. In this section we show that one can easily check unordered codes under that assumption. In section 5.5 it will be demonstrated that one cannot improve the handling of those codes under that same assumption.

It is possible to use m-out-of-n or Berger codes in sections of the computer different from memory or busses and where the single fault model is generally accepted. The trade-off is between translation into a more appropriate code and having to care for unnecessary redundancy. Of course, if the fault model is no longer unidirectional, as typically it won't be in the case of checkers, then it is possible to relax the inverter-freeness constraint on the design. Moreover the improbable fault set is increased to $\mathfrak{I}_{uni} - \mathfrak{I}_{single}$. For example for k-out-of-2k codes under the single fault assumption one has to check only for non-codewords of the form (k-1) and (k+1)-out-of-2k. This means that parity checking is sufficient.

Indeed in both the fixed weight codes and the Berger codes, the parity bit is evident. In fixed weight codes, the parity bit (or its complement) is just any bit. In Berger codes the least significant check bit is the parity of the information bits. Checking can be accomplished simply by using a TSCC for parity codes. These two situations are described in Figures 3.2a and b. Also note that in the case of Berger codes, translation into parity codes is accomplished by simply ignoring all the check bits except the least significant one. Figure 3.2c shows an example where this idea is used to advantage. Code translation will be discussed further in Chapter 5 and the parity checked adder in Chapter 4.

Smith's checker (Figure 3.3) is an excellent checker which is TSC under the unidirectional fault assumption. However it is too expensive to

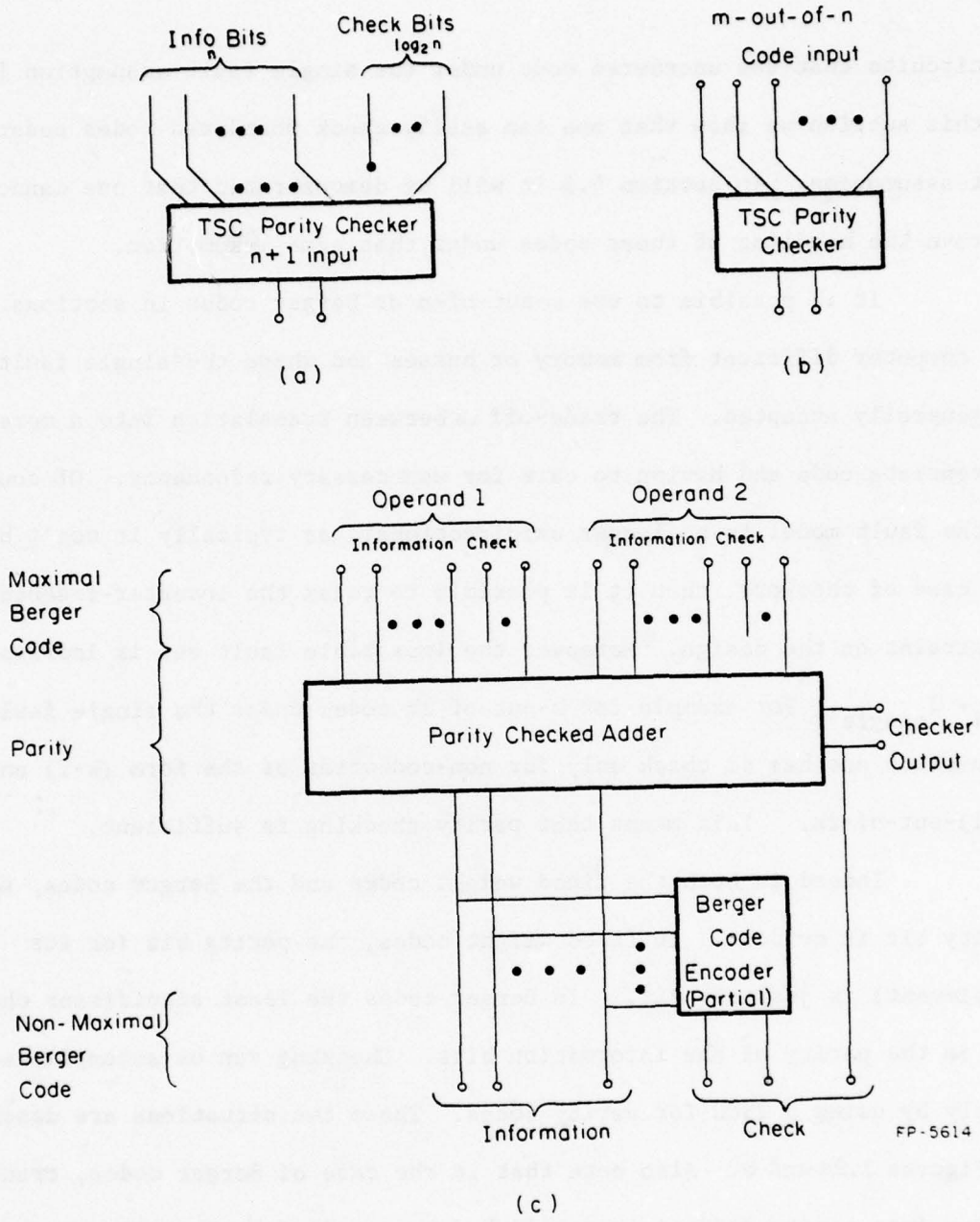


Figure 3.2. Checkers for unordered codes under the single fault assumption (a and b) and an application to addition (c).

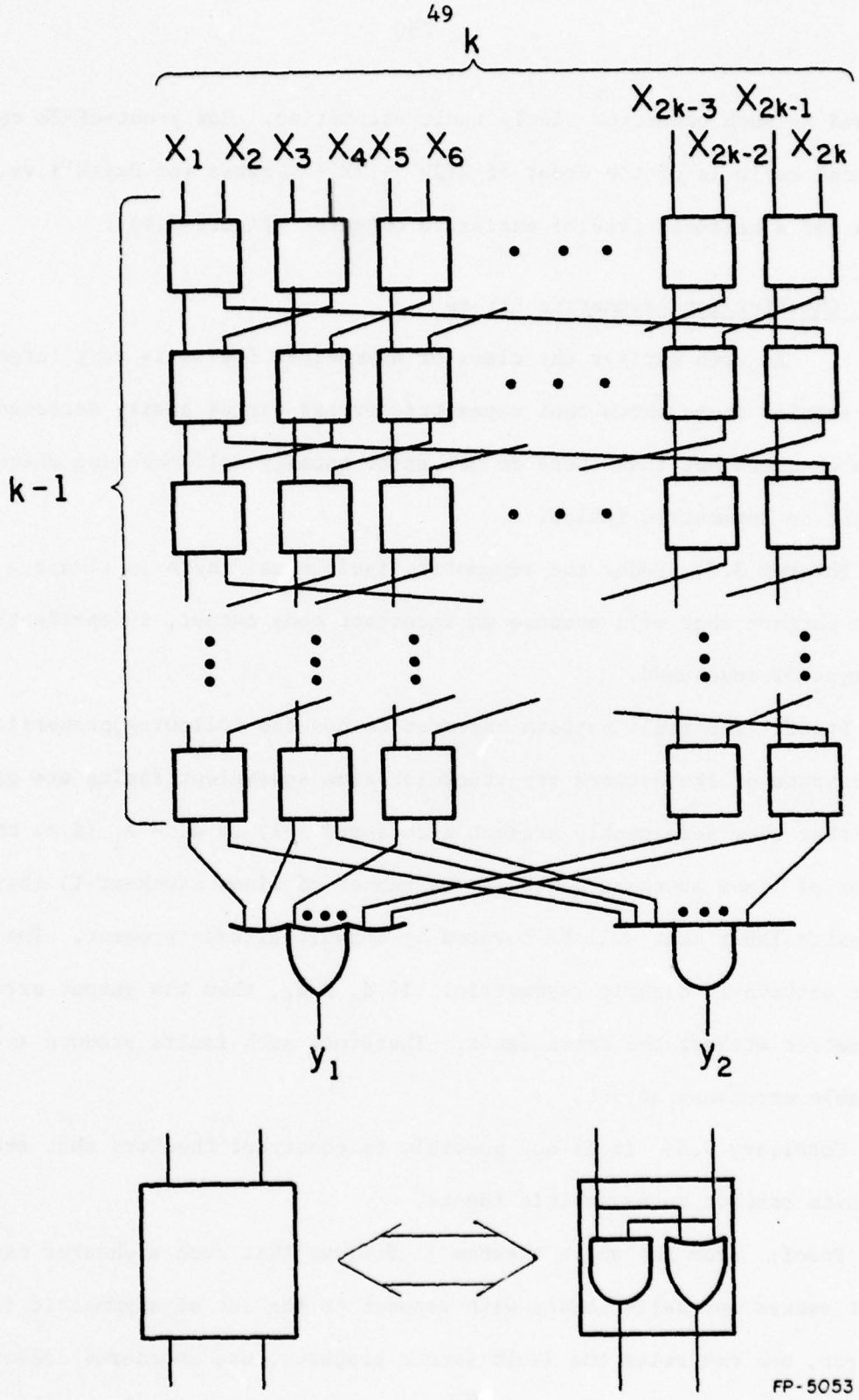


Figure 3.3. Smith's TSC checker for k-out-of-2k codes [51].

be used as such under the single fault assumption. For k-out-of-2k codes the cost ratio is of the order of $k(2k^2 + 2k + 2)$ gates for Smith's vs. $2(k-1)$ gates for a balanced tree of exclusive OR gates (Figure 3.4)).

3.8. Checking for Asymmetric Errors

As seen earlier the class of asymmetric faults is very large. In this section it is shown that asymmetric errors can be easily detected using known checkers but that there do not exist totally self-checking checkers with respect to asymmetric faults.

Theorem 3.7: Under the asymmetric fault model there is always a possible fault pattern that will produce an incorrect code output, independently of the type of code used.

Proof: The fault pattern referred to has the following properties: 1) The outputs of the network are stuck (or some equivalent faults are present) such that they permanently present a codeword. 2) If $d_0 = d_1$ (i.e. the number of lines stuck-at-0 equals the number of lines stuck-at-1) then insert one extra fault that will be covered by a fault already present. The resulting fault pattern is clearly asymmetric. If $d_0 \neq d_1$, then the output error is asymmetric without the extra fault. Therefore such faults produce an undetectable erroneous output. Q.E.D.

Corollary 3.8: It is not possible to construct checkers that are TSC with respect to asymmetric faults.

Proof: From the above theorem it follows that such a checker cannot be fault secure nor self-testing with respect to the set of asymmetric faults. However, one can relax the fault secure property, use unordered codewords and produce a checker that will detect all asymmetric errors.

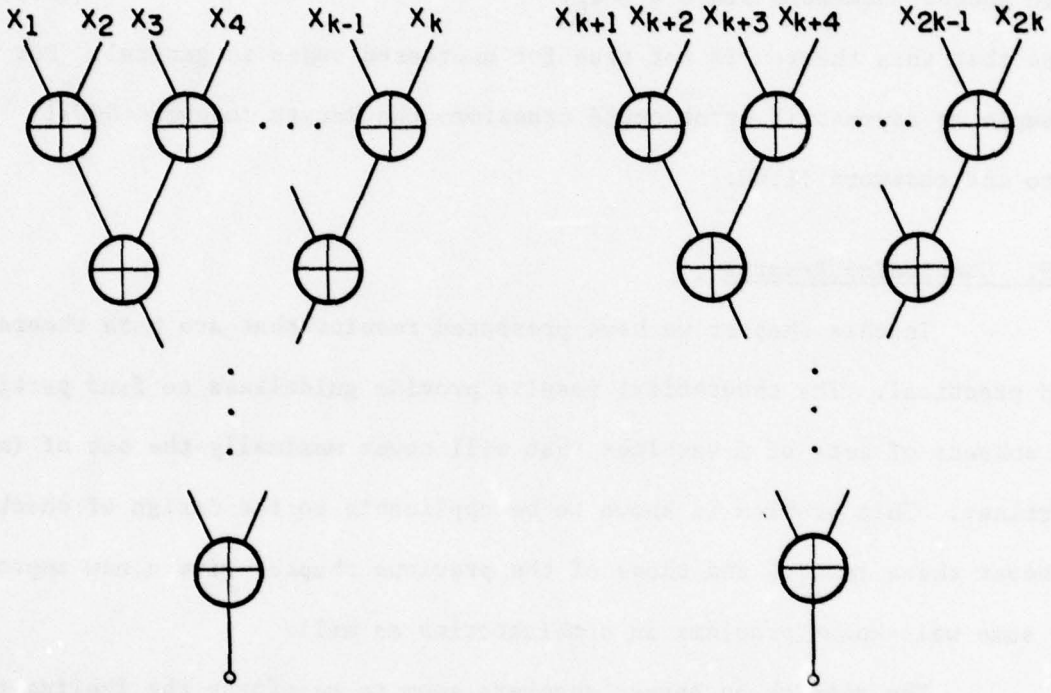


Figure 3.4. Balanced TSC checker under the single fault assumption for k-out-of-2k codes.

Theorem 3.9: A totally self-checking checker under unidirectional faults is also asymmetric error preserving if the code spaces are fixed weight code spaces.

Proof: Any asymmetric error with d_1 and d_0 can be viewed as a unidirectional error of multiplicity $|d_1-d_0|$ and direction 1 if $d_1 > d_0$ and direction 0 otherwise. Moreover a fixed weight code will never be changed into another codeword since $d_1 \neq d_0$. Q.E.D.

Note that this theorem is not true for unordered codes in general. For example an asymmetric error could transform the Berger codeword 00011 into the codeword 11100.

3.9. Concluding Remarks

In this chapter we have presented results that are both theoretical and practical. The theoretical results provide guidelines to find partitions or subsets of sets of m-vertices that will cover maximally the set of (m-1)-vertices. This problem is known to be applicable to the design of checkers. However these results and those of the previous chapter give a new approach to some well-known problems in combinatorics as well.

The results on Berger checkers seem to reinforce the feeling that the price to pay for separability of information and redundancy lies in more difficult or less appealing checkers.

On the practical side we have seen some compromises that result in substantial savings in designing checkers for unordered codes under the single fault assumption. Also fixed-weight codes have been shown to be superior in error coverage, an advantage in memories where faults and errors coincide generally. There the cost is reflected in the circuitry needed to extract

meaningful information. This problem is going to be discussed in Chapter 5, after another fault model and circuit structures are discussed in Chapter 4.

The reason for looking at arithmetic faults and codes is simply that unordered codes are not easily adaptable to arithmetic circuits. It appears that each code is ideal under a particular fault model. It is proposed that different codes be used where they fit well hence the need to look at arithmetic codes for designing TSC ALUs. This is done in the next chapter. Chapter 5 will discuss the general problem of going from one code to another.

4. TOTALLY SELF-CHECKING ARITHMETIC CIRCUITS

4.1. Introduction

In this chapter we shall be concerned with the checking of arithmetic operations, using well-known arithmetic codes: parity, AN and residue codes. Fixed-weight codes, like the biquinary and 2-out-of-5 codes, have been used in decimal adders and an extension to general adders is far from simple [48]. Ho has studied a TSC two-rail adder and incorporated it in a TSC system [24]. Algebraic codes are mainly communication codes and have been mostly used only as such. One exception to this is the possible use of Reed-Muller codes to check (and correct) addition; it necessitates, however, a special adder structure [41].

Arithmetic codes are designed to correct or detect arithmetic errors i.e. errors that can cause borrow and carry propagation of logic faults resulting in several faulty output bits. These errors are characterized by output changes of the form $\pm 2^i \pm 2^j \pm \ldots$. Techniques to check parity of words are simple and well-known. However checking arithmetic using parity codes requires access to internal carries of the adder and duplication of carries is required for increased error coverage. As expected complexity of checking increases with increases in concurrency and look-ahead. On the other hand residue checking requires no modification to the adder. The check bits of the sum are obtained only from the check bits of each operand by simple modular addition. More complex adders may impose more check bits on the operands but the design procedure is always straightforward.

Similar observations can be made about AN codes. Kolupaev presented self-testing residue trees [28]. His approach combines fixed-weight code and residue codes in the sense that these trees were mapping residue codes

into codes of the form 1-out-of-2^n where n was the byte size. Although modular (2 types of module were used) this approach exhibits very large fan-in and is generally very expensive as well. Our approach shall be more conventional.

4.2. Arithmetic Codes

This section presents a brief overview of arithmetic codes. Parity codes will also be dealt with, but need little introduction.

Definition 4.1: An expression for an integer I of the form $I = \sum_{j=0}^{n-1} i_j \cdot 2^j$, $i_j \in \{-1,0,+1\}$ is called the modified binary form (MBF) of I.

There exists a "canonical" MBF for each integer; it has the properties that no two non-zero i_j's are adjacent (also called "non-adjacent form") and that it is unique.

Definition 4.2: The arithmetic weight of the integer I, denoted $W_A(I)$, is the least number of non-zero coefficients which are necessary in a MBF of I.

Definition 4.3: The arithmetic distance between integers I_1 and I_2, denoted $D_A(I_1,I_2)$ is the arithmetic weight of their difference i.e. $D_A(I_1,I_2) = W_A(I_1-I_2)$.

Definition 4.4: An AN code can be defined as a code generated by a positive integer A and is the set of integers AN for $0 \leq N \leq B$, where B is specified.

It can easily be verified that the sum and product of AN codepoints are also AN codepoints. The code capabilities of AN codes are defined in the following theorem:

Theorem 4.1: For any $t \geq 0$ and $s \geq 0$, an AN code can correct all errors of weight t or less in its codepoints and can detect all other errors of weight t + s or less in its codepoints if and only if $D_{A\,min} > 2t + s$.

Definition 4.5: Let Z_m be the set of integers modulo m. A multiresidue code in Z_m, generated by the set of integers $m_1, m_2, \ldots, m_k$ such that $m_i > 0$ and m_i divides m, is the set of (k+1)-tuples of integers

$$N = \{N, [N]_{m_1}, [N]_{m_2}, \ldots, [N]_{m_k}\} \text{ where } [N]_{m_i} = N \bmod m_i$$

Multiresidue codes with a single residue check will be called simply residue codes. Again it can easily be seen that the check digits of a sum or a product of 2 encoded numbers are the sum or product of the check digits of the encoded numbers. These codes are also clearly systematic.

Definition 4.6: The associated AN code for a multiresidue code is the AN code whose generator is A = least common multiple (m_i, i = 1,k) and whose number of codepoints B satisfies m = AB.

The code capabilities can now be defined in the following theorem:

Theorem 4.2: A multiresidue code has the same error detecting and correcting abilities as its associated AN codes (Proof in [42]).

A final comment is in order. Although minimum arithmetic distance is used to define code capabilities in an arithmetic environment, codes with similar minimum distance may have completely different overall error coverage. For example the residue 3 and 15 codes will both detect all single arithmetic errors but while only 2/3 of all possible error patterns (in a 4-bit byte) are detected by the residue 3 code, the residue 15 code will detect 14/15 of all possible error patterns. This will be discussed further in section 4.3.

4.3. Comments on the Choice of a Modulus

In theory any modulus or any generator can be used to construct arithmetic codes. In fact moduli of the form $\pm 2^i \pm 2^j \pm 2^k$ (like 11, 13, 19, 21 etc.) can be attractive since they have a minimum arithmetic distance

equal to 3, hereby providing double arithmetic error detection or single arithmetic error correction. However it is relatively difficult to check those codes as they are awkward to check using regular binary arithmetic. One has either to provide additional logic to conventional adders or to design a new adder with the proper modulus so they can be used as checkers. In any case the resulting checker will have very little structure since the weight distribution of the bits in a binary encoded argument, modulo a relatively prime modulus, has an order equal to the modulus minus one. In other words the size of the bytes equals the modulus-1, which is inconvenient as usual word sizes are multiples of 8 or 12. These difficulties have prevented generalized use of such moduli.

The most common moduli are of the form 2^n-1. Although these moduli provide only single arithmetic error detection (their "real" capabilities are discussed in the next section) checking is accomplished by simple one's complement addition. Moreover the one's complement algorithm applies directly to left and right shiftings, complementation and other more specialized operations [33]. Encodings and decoding into/from non-separate codes is also easy. Avizienis discusses the properties and details of implementation of such codes [4,6]. The design of checkers for (2^n-1)-encoded arguments is straightforward but it results in slow operation due to end-around carries. So far it is the only practical approach that has been used for arithmetic codes.

4.4. Different Concepts of Distance

The capabilities of arithmetic codes are defined in terms of their arithmetic or modular distances. On the other hand the inputs to a checker have code capabilities which are usually best measured in terms of Hamming distances. In this section we will look at some of the considerations imposed by this discrepancy.

Definition 4.7: The circuit distances of a circuit is the set of Hamming weights of possible errors at the circuit outputs caused by a modeled fault for code space inputs.

The bound of interest in the case of circuit distance is the upper one: the maximum circuit distance. Clearly we have:

1) A circuit is fault-secure if the maximum circuit distance is less than the minimum Hamming distance of the output code. (In general errors will be detected if the code distance set is disjoint from the circuit distance set) [1].
2) In a bit sliced circuit the maximum circuit distance cannot exceed the multiplicity of the largest modeled fault.
3) The maximum circuit distance is bound by the number of outputs.

As indicated previously the relation between arithmetic distance and Hamming distance (which is in turn simply related to circuit distance) is not straightforward: the Hamming distance in arithmetic codes is a function of the codewords themselves as well as the set of arithmetic distances. The Hamming distances between arithmetic codewords vary much more widely than the Hamming distances of good (well-packed) communication codes. However bounds on both types of distance are simply related and sufficient conditions can be derived from those to design totally self-checking arithmetic checkers. In the following we investigate briefly this relationship and shall provide guidelines to design functional circuits that have maximum circuit distance sufficiently small for given arithmetic codes.

Example 4.1: The example illustrates the effects of single faults on the value of the outputs of a full-adder (Figure 4.1). The faults on the left of the dotted line may affect both the sum and the carry while those on the right will not.

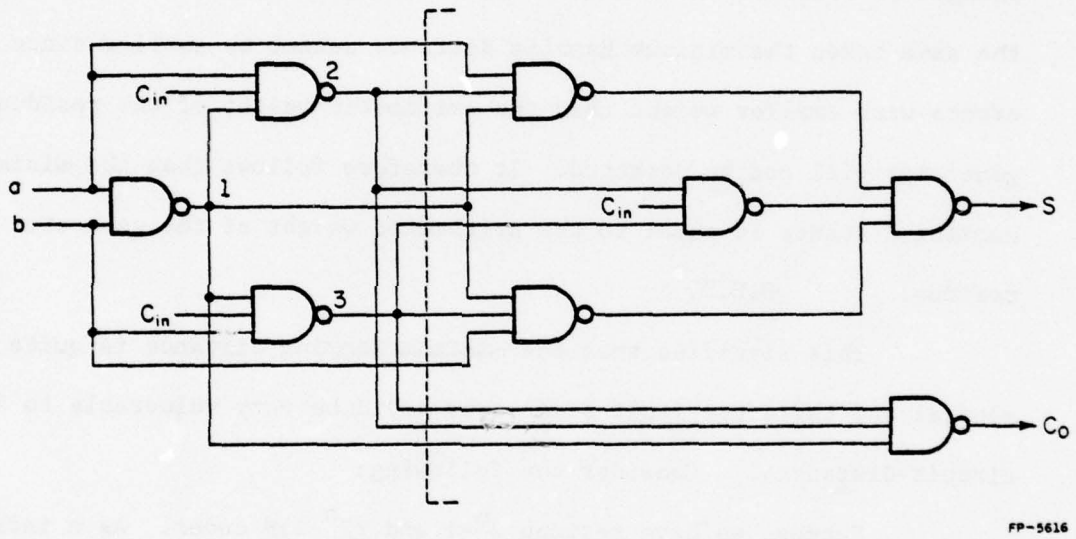


Figure 4.1. A full-adder.

Input			No-Fault output		Faulty output		Fault	Output change
a	b	C_{in}	S	C_0	S	C_0		
1	1	0	0	1	1	0	line 1	$+2^i - 2^{i+1}$
1	1	1	1	1	0	1	stuck-at-1 and equivalent	-2^i
1	0	1	0	1	1	0	line 2	$+2^i - 2^{i+1}$
							stuck-at-1 and equivalent	-2^i
0	1	1	0	1	1	0	line 3 stuck-at-1 and equivalent	$+2^i - 2^{i+1}$

Theorem 4.3: Arithmetic codes (AN or Residue) have a minimum Hamming distance equal to the arithmetic weight of the generator or residue.

Proof: The minimum Hamming distance cannot be greater since a carryless or borrowless addition of an arithmetic error vector equal to the generator or residue results in another codeword; that is in an undetectable erroneous output. In other words there exist arithmetic error vectors of arithmetic weight equal to that of the residue or generator that cause Hamming weight changes at least equal to the arithmetic weight which are undetectable. By the same token the minimum Hamming distance cannot be smaller since some errors with smaller weight than the arithmetic weight of the residue or the generator will not be detected. It therefore follows that the minimum Hamming distance is equal to the arithmetic weight of the generator or residue. Q.E.D.

This signifies that the minimum Hamming distance is quite small in general and therefore fault secureness could be very vulnerable to larger circuit-distances. Consider the following:

Suppose we have residue 2^n-1 and $(2^n$-1)N codes. As n increases the redundancy increases. However, according to the above theorem, no matter how large n is, codes of the form 2^n-1 are equally vulnerable. Since the theorem states that there is a double arithmetic fault, namely 2^n-1 itself, that will not be detected. In practice, however, one has to look hard to find a single fault in the adder circuitry that produces such a double fault (for n large). The overall code capabilities actually increase and it may be justified since 2^n-2 error patterns are now detectable. Indeed the picture is not so bleak when the distribution of Hamming weight changes, at the output, is considered. This is formalized in the concept of arithmetic circuit distance set.

Definition 4.8: The arithmetic circuit distance set is the set of arithmetic weights of potential error patterns caused at the output by members of the fault set.

Of special concern are those patterns that can be generated at the end of paths that originated from fanout points. They have the largest arithmetic weights and are the most likely to produce changes at the ouputs that are equal to the modulus or the generator.

Theorem 4.4: In an arithmetic circuit if the arithmetic weight of the generator or the modulus is not a member of the arithmetic circuit distance set, then the circuit is fault secure.

Proof: It follows directly from the definition of fault secureness and the previous discussion.

4.5. Arithmetic Codes as Test Sets

In this section we show that common arithmetic codes provide a sufficient test set for adders that are fault secure with respect to the types of arithmetic errors that the codes are designed to protect against. It will first be indicated that the AN codes considered are cyclic, that there exists a test set for a small number of consecutive full-adders and finally show that it covers all adders in a parallel adder using the cycling property.

Theorem 4.5: The generator A and the number of codewords B in a cyclic AN-code satisfy $AB = 2^j-1$. Conversely, every A which divides 2^j-1 generates a cyclic code with $B = (2^j-1)/A$ codewords (statement and proof in [30]).

This theorem seems to impose a constraint on the size of the code. Previously, the only requirement was that the generator be of the form 2^n-1. these two apparently conflicting cases can be reconciled by noting that the constraint $AB = 2^j-1$ simply implies that the codeword size be an integer

multiple of the byte size. For example, remarking that $3 = 2^2-1$ divides 2^j-1 if and only if j is even, it can be seen that codeword sizes should also be even, i.e. of length 2,4,6 etc. Similar consideration can be applied to other moduli and is formalized in the theory of exponents [31].

To test an irredundant adder it is sufficient to apply all possible input combinations. Because of carry dependencies, more than one adder stage has to be considered. For ripple-through carry adders, 2 stages are sufficient; it may be more for adders with look-ahead. Because of the cycling property, this test set will be eventually applied to all stages. For 2 stages, assuming bytes of size n (i.e. generator = 2^n-1) all the combinations are produced by the first 4 codewords. Underlined bits are those applied to the first 2 stages.

	$\overbrace{}^{n}$	
0	$\underline{00...00}$	
2^n-1	$11...\underline{11}$	
$2(2^n-1)$	$111...\underline{10}$	
$3(2^n-1)$	$11...\underline{01}$	$= (2^n-1) + 2(2^n-1)$

Each operand will eventually produce these codewords independently so that all 16 input combinations will be applied. The cycling property will then insure that all stages will be checked for carry and sum errors. For adders with a more parallel structure this analysis must be repeated.

For residue codes the problem is much simpler. The adder and the checker are completely independent units and both will receive all possible input combinations and are therefore totally self-checked.

4.6. Checking Arithmetic Codes

In this section, we show that, under certain conditions, essentially all known methods for checking arithmetic can be extended easily so that they are totally self-checking. We shall see that the main difference lie in the size of the hardcore. First, a look at parity codes:

In the case of parity checking, the main constraints are imposed on the adder itself. The parity P_S of the sum ($S = A + B$) can be computed as follow:

$P_S = P_A + P_B + P_C$ where P_C is the parity of the carries.

However, in practice a fault may affect both the carry and the sum digits, producing an undetected error. One approach to the problem is to duplicate the carries i.e. generate independently pairs of carries. One set of carries is then used to compute the sum and the other set is used for checking. The adder must also have a small (= 1) or odd circuit distance so that it is fault-secure. The parity of the sum is then predicted by computing the modulo 2 sum of the parity of the operands and of the carries. The result of the sum (or subtraction) along with its predicted carry are then fed into a totally self-checking parity checker (Carter and Schneider [11], Anderson [1]). It is possible, although more expensive, to merge prediction and checking into the double tree of exclusive-ORs forming the TSC checker.

We now turn our attention to arithmetic codes of the form $(2^n-1)N$ or residue (2^n-1). Consider Figures 4.2 and 4.3 describing serial and tree structured checking. In the first instance checking is accomplished in a serial fashion: bytes of length n are taken one at the time and accumulated modulo 2^n-1 (i.e. modulo the residue or the generator). The overall delay is, of course, the number of bytes times the assimilation time in the checker.

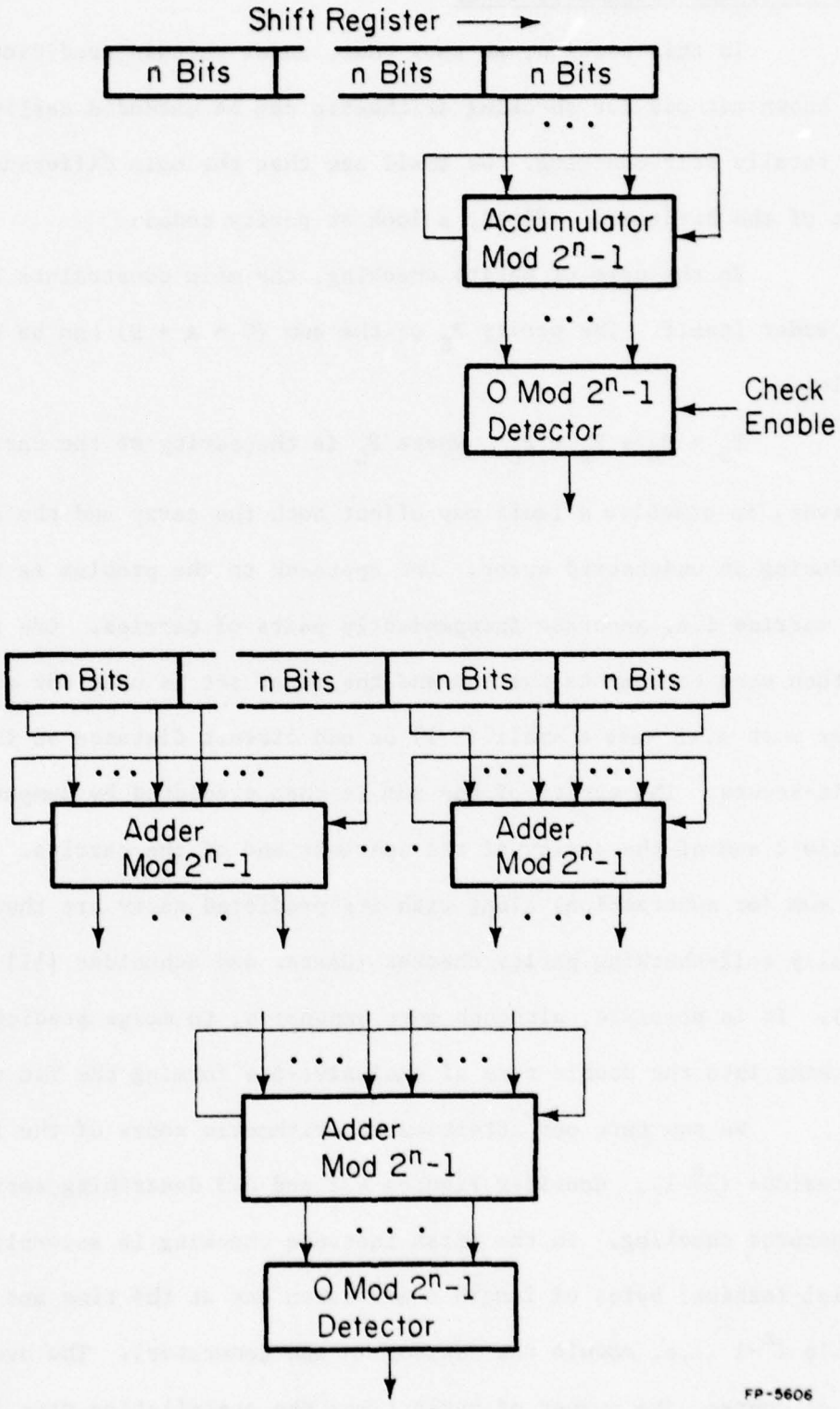


Figure 4.2. Serial and tree checkers for AN codes.

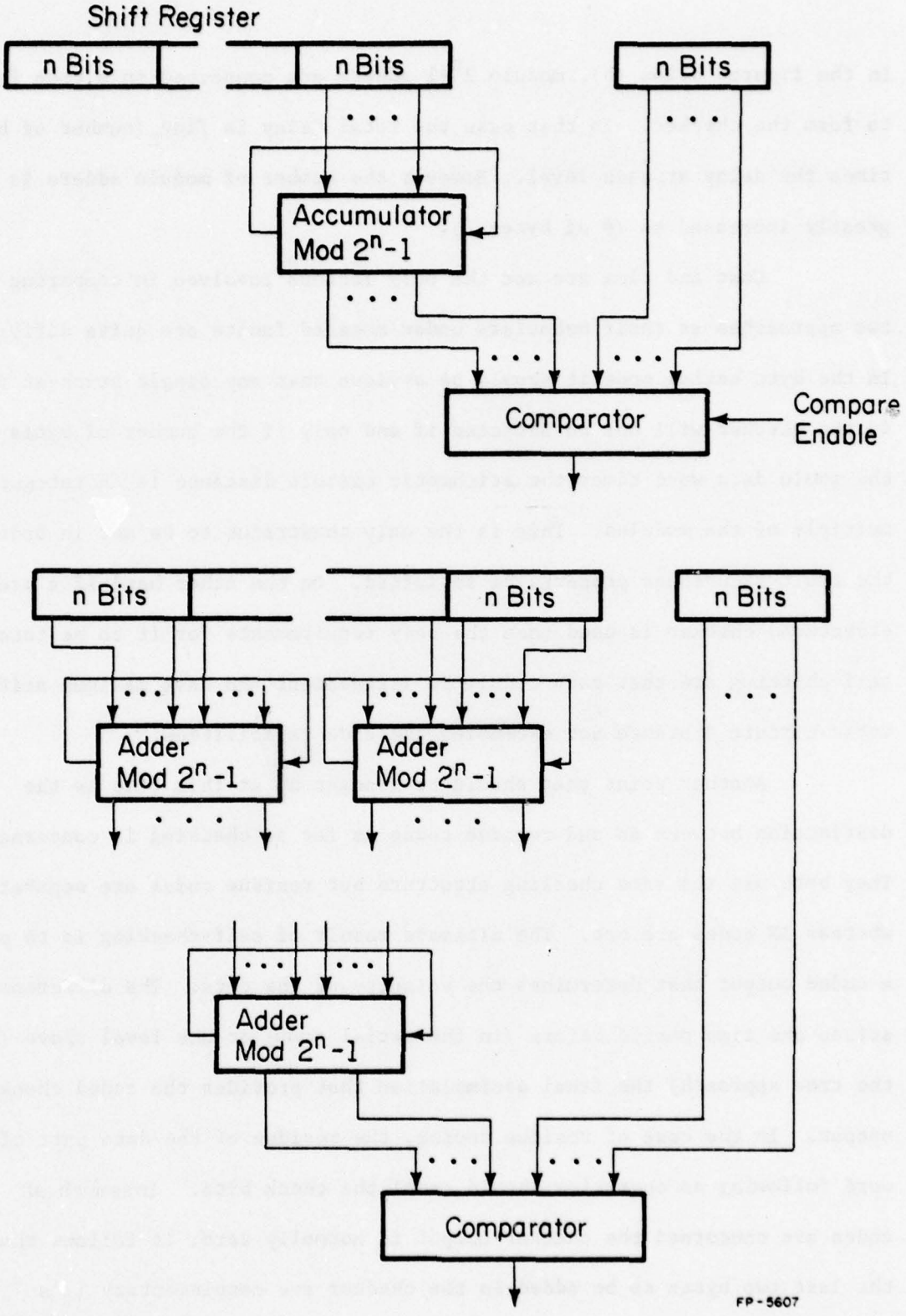


Figure 4.3. Serial and tree checkers for residue codes.

In the figures below (b), modulo 2^n-1 adders are connected in a tree fashion to form the checker. In that case the total delay is $\lceil \log_2 (\text{number of bytes}) \rceil$ times the delay at each level. However the number of modulo adders is greatly increased to (# of bytes-1).

Cost and time are not the only factors involved in comparing the two approaches as their behaviors under modeled faults are quite different. In the byte serial mode it should be obvious that any single stuck-at fault in the checker will not be detected if and only if the number of bytes in the whole data word times the arithmetic circuit distance is an integer multiple of the modulus. This is the only constraint to be met in order that the fault-secureness property be satisfied. On the other hand if a tree-structured checker is used then the only requirements for it to be totally self-checking are that each module be irredundant and have maximum arithmetic circuit distance not exceeding the code capabilities.

Another point that should be brought up at this time is the distinction between AN and residue codes as far as checking is concerned. They both use the same checking structure but residue codes are separate whereas AN codes are not. The ultimate result of self-checking is to provide a coded output that determines the validity of the data. The difference arises one time period before (in the serial mode) or one level above (in the tree approach) the final assimilation that provides the coded checker output. In the case of residue coding, the residue of the data part of a word following an operation should equal the check bits. Inasmuch AN codes are concerned the checker output is normally zero: it follows that the last two bytes to be added in the checker are complementary (1's complement) or all zeros. The all-one vector may normally never be produced in the hardware even thought is is correct. Also no matter how 0 mod 2^n-1

is encoded, the checkers described before will produce the all-zero vector if and only if they are presented with all-zero inputs. From the previous considerations it should be clear that due to dependencies at the final level, the checker may not operate in a fault-secure mode at that instant and may not be completely tested. In the case of serial checking the adder in the final assimilation period is neither operating in a fault secure nor self-testing mode. However it may not matter since the probability that conditions propitious to the generation of an erroneous code output at that time are virtually nil. It may however be unsatisfactory from a rigorous point of view on self-checking. For tree-checking the problem arises also but for both AN and residue codes where only equal or complementary inputs plus the all-zero input are available. The problem is now more acute since the final module is never fault-secure and is not completely self-tested.

For AN codes the problem may be alleviated by replacing the final adder by a row of exclusive-OR gates that compare corresponding bits from each of the two final bytes. This leads to the situation where the hardcore will be larger than usual. The number of lines that will form the hardcore will be equal to the byte's size and will take the value of all zeros or all ones. Clearly further reduction is not possible. It should also be emphasized that the TSC property of this final level of exclusive-OR gates is conditional on the sufficiently repeated application of the all-zero or all-one vector to its inputs so that ouputs can alternate and therefore be checked. As mentioned earlier the checker will produce the all-zero vector if an only if it is presented with the all-zero vector. In a computer the all-zero vector, being an initialization vector, is certainly more common than any other vector so it is reasonable to assume that the level of exclusive-or gates is going to be sufficiently tested. Let us summarize and contrast with residue codes:

For AN codes: The inputs to the final assimilation are of the form

$$x_1x_2\ldots x_n\ \bar{x}_1\bar{x}_2\ldots\bar{x}_n \text{ and hopefully } \underbrace{0\ldots0}_{n}\ \underbrace{0\ldots0}_{n}$$

-The final level of exclusive-OR gates is totally self-checked

-The hardcore consists of n lines, the all-one or all-zero vector indicating a correct codeword.

For residue codes: The inputs to the final assimilation are of the form

$$x_1x_2\ldots x_n\ x_1x_2\ldots x_n$$

-A final level of exclusive-OR gates will not be totally self-checked.

-The hardcore therefore consists of 2n lines.

So for residue codes (and to a lesser extent for AN codes) we are faced with a fundamental problem, namely a substantially sized hardcore. It is not solved by merely inverting one byte and feeding it to a level of exclusive-OR gates (which is the same as using inverse residue coding [4]). In doing so, one would still have to generate $\underbrace{0\ldots0}_{n}\ \underbrace{1\ldots1}_{n}$ or $\underbrace{1\ldots1}_{n}\ \underbrace{0\ldots0}_{n}$ which are certainly valid codewords; they are also never normally generated. Possibly the simplest way around is to devise some scheme that is going to generate periodically vectors of the form 0^n1^n or 1^n0^n. This may be equivalent to moving the problem into another section of the ALU where those codewords may be generated by a unit that is perhaps not totally self-checked. Another approach would be to design from scratch checkers with the desired properties. So far no general method that eclipses the simplicity and structure of checkers constructed out of modulo 2^n-1 adders, has been found.

There are other problems that must be looked at when designing TSC arithmetic checkers. For example special care must be exercised in two cases:

1) Design of checkers for minimally redundant arithmetic codes.

2) Design of checkers for arithmetic circuits with large amount of look-ahead or parallelism (i.e. circuits with large arithmetic circuit distance).

Clearly most designs will fall between these 2 extremes, so that guidelines that take into account each of these will always apply, but to varying extent. In the case of minimally redundant arithmetic codes, i.e. the residue 3 and 3N codes, precautions must be taken in designing both the adder and the checker so that no error will produce output changes equal to the modulus. This is the case when any two adjacent output bits are erroneous unidirectionally. Incidentally "standard" checkers for these codes are totally self-checking with just 2 outputs. This is discussed further in the example of the next section. Fortunately for codes with identical arithmetic distance but with more redundancy this problem does not exist: any changes in boundary bits of different bytes will never cause a change equal to the modulus. However, carry propagation schemes other than the basic ripple through carry might possess just the right defeating arithmetic circuit distance in the carry propagation circuit. So as in the case of minimally redundant codes care must be taken so that circuits with large arithmetic distance do not exceed the code capabilities.

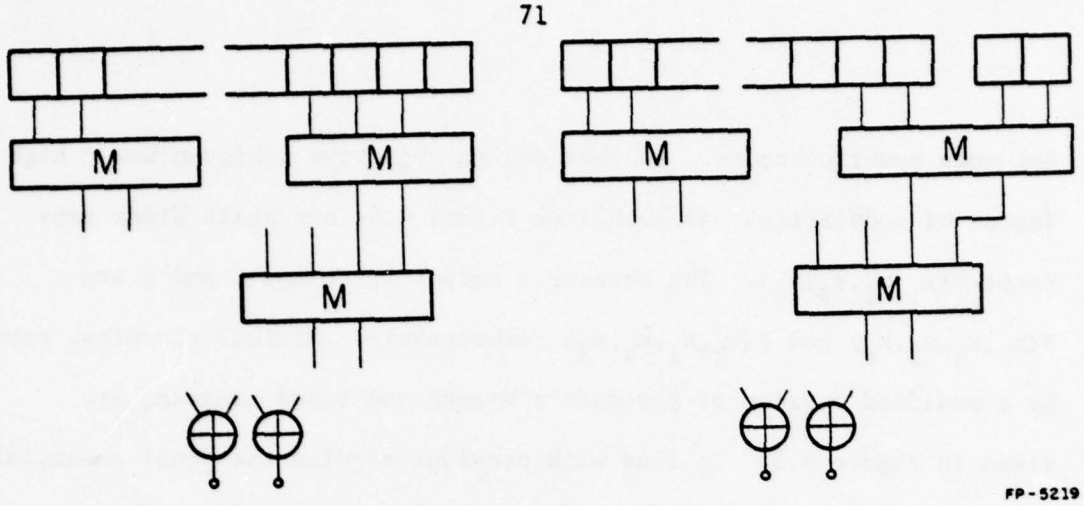
4.7. A Low-cost TSC Checker for Residue 3 and 3N Codes [16].

This section illustrates the kind of considerations one has to make as well as the type of results that would be satisfactory in designing arithmetic checkers. The example used is the design of a low-cost totally self-

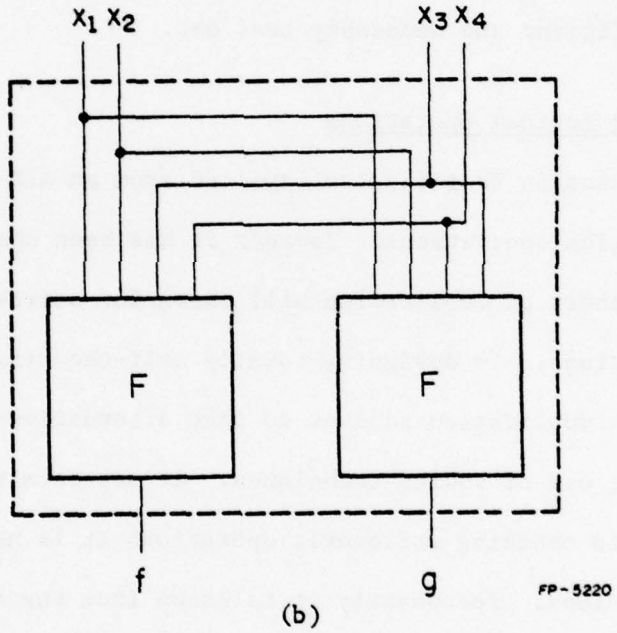
checkers for the residue 3 and 3N codes. These codes are the least redundant arithmetic codes. It is therefore assumed that the adder has a circuit distance of at most 1. One sufficient (but not necessary) condition for this situation to be achieved is to require that the adder produces independent sum and carries. Such a circuit in its minimal version requires 9 gates per stage versus the absolute minimum of 6. Other minimal adders have a circuit distance of 1 and the only way to find their circuit distance is to examine their behaviour under single faults.

The purpose of the design is to find a functional block that can be used as in Figure 4.3 or that can be interconnected as in Figure 4.4 to produce ultimately the checker's outputs. In Figure 4.4 the checker is connected in a tree fashion to the 3N-encoded adder's output. A 3N-coded result causes the final output of the checker to be either 00 or 11. In the case of residue 3 coding the situation is essentially the same except that the check bits are separate from the data word. In both cases it does not matter at which level of the checker the inputs are connected as long as the overall structure is acyclic and the weight order respected (i.e. lines with weights 1 and 2 are connected to corresponding inputs of the modules).

In the course of the design it has been possible to obtain an encoding that permitted a more uniformly distributed alternation of the output signals. The encodings for 0 mod 3 were allowed to be both 00 and 11. This is actually a secondary result, as the prime target was to introduce more symmetry in the function realizing the checker. Another design goal was the elimination of end-around carries for a faster operation: the time of assimilation is reduced by half at each level. It is easily seen that this property will probably be the most difficult one to generalize



(a)



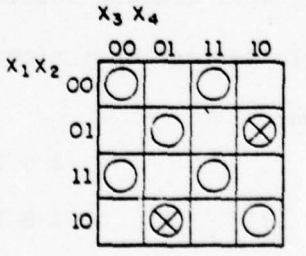
(b)

f

$x_1 x_2$ \ $x_3 x_4$	00	01	11	10
00	1	0	1	1
01	0	1	0	0
11	1	0	1	1
10	1	0	1	0

g

$x_1 x_2$ \ $x_3 x_4$	00	01	11	10
00	1	1	1	0
01	1	0	1	0
11	1	1	1	0
10	0	0	0	1



○ Code Space ⊗ Improbable Code Space

FP-5221

(c)

Figure 4.4. A low cost tree checker for residue 3 and 3N codes.

for more powerful codes. Another design objective achieved was a high degree of modularity. As seen from Figure 4.4, one basic block performs $F(x_1,x_2,x_3,x_4)$. The checker's output functions f and g are $F(x_1,x_2,x_3,x_4)$ and $F(x_2,x_1,x_4,x_3)$ respectively. Minimal circuits, computed by a modified version of Davidson's branch and bound program, are given in Figure 4.5. In line with previous results the final assimilation is performed using exclusive-OR gates which are TSC. The code must however be produced by the same type of circuits (i.e. the same f and g) so as to provide a sufficient and necessary test set.

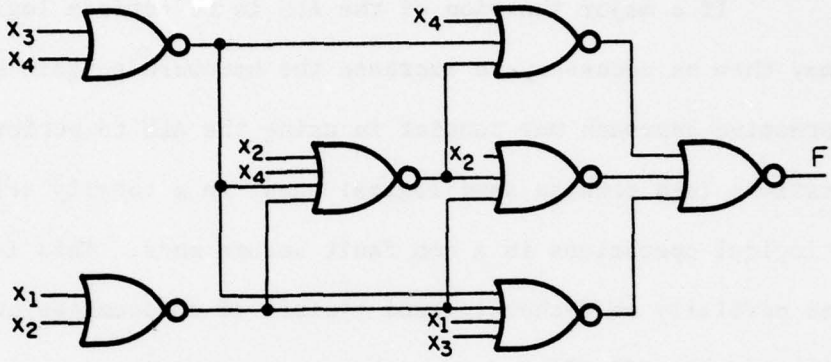
4.8. Checking Logical Operations

A function that is also expected from an ALU is the capability to perform logical operations. However it has been shown by Peterson [33] that nothing short of duplication will check for correctness of bitwise logical operations. In designing totally self-checking circuits one is competing with duplication schemes so that alternatives have to be found to justify the use of coding techniques. It may seem that no matter what code is used in checking arithmetic operations it is not going to work for logical operations. Fortunately it is known from the days of ALU's with very limited logical powers that all logical operations can be performed using well defined sequences of addition (subtraction), shifting and a single Boolean operation. For example the following relations are well known:

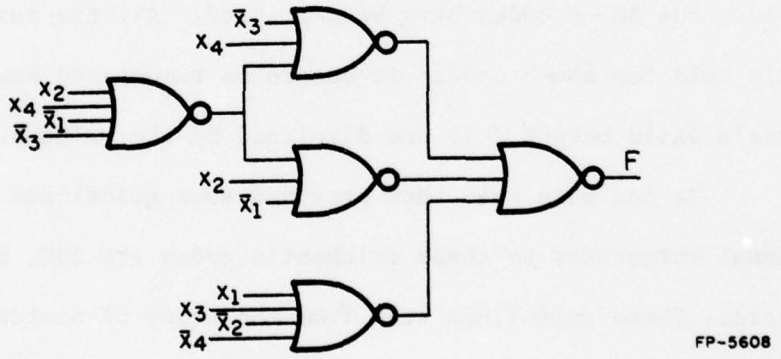
$$A + B = A \wedge B + A \vee B$$

$$A \oplus B = A + B - 2(A \wedge B)$$

where logical operations are performed bitwise. Therefore, given the added cost of an AND circuit it is possible to compute the bitwise OR



(a) Uncomplemented variables only



(b) Complements available

Figure 4.5. Minimal NOR implementation of the checker.

and exclusive-OR of two words using simple arithmetic. One cannot escape the fact that at least this AND section must be duplicated in order for the whole structure to be totally self-checking. This AND circuit must be checked by a TSC duplication comparator [1].

If a major function of the ALU is to perform logical operations it may then be necessary to increase the hardware to gain some speed. An alternative approach may consist in using the ALU to perform arithmetic operations (and perhaps somd logical ones) in a totally seif-checking mode and logical operations in a non fault secure mode. This is often referred to as partially self-checking and typical of compromises currently made in implementation [56,57].

4.9. Concluding Remarks

Throughout this chapter, references to the more general case of AN codes, the AN + B codes have been omitted. All the results for AN codes clearly hold for AN + B codes; it has to be remembered however, that the checker's valid output 0 is now displaced by the value B.

It has been seen that provided some guidelines are followed conventional structures to check arithmetic codes are TSC, but have a larger hardcore. These guidelines stem from the study of distance and its relation to arithmetic distance. Because the byte's size of residue 3 and 3N codes coincides with the smallest possible hardcore it was possible to design a checker for such codes without relying on modulo 3 adders. This example also pointed out some desirable characteristics that less conventional TSC checkers for arithmetic codes should ideally have.

Arithmetic codes have been seen to lend themselves naturally to the design of TSC arithmetic circuits. As observed in the discussion on distance their error coverage is best expressed in terms of arithmetic distance and when used elsewhere in a computer these codes have an error coverage which is difficult to define accurately in terms of the dominant fault model. In other words arithmetic codes generally perform well only in arithmetic circuits and other codes should be used in the non-arithmetic sections of a computer. The following chapter presents general considerations concerning the use of different codes in a single digital system.

5. CODE TRANSLATION

5.1. Introduction: A Summary of the Advantages of Codes

In the preceding chapters codes were presented along with the fault models they were to protect against. Parity codes are designed to protect against single errors. In general they will detect any asymmetric error of odd net weight. It is difficult to design circuits so that this will always be the case. Therefore parity checking is usually restricted to single fault detection in logic circuits. Parity codes can also be used in arithmetic circuits. It requires internal modification to the adders but it is in general acceptable. The next codes in terms of redundancy are the least redundant arithmetic codes followed by all other such codes. As indicated earlier the simplest type of arithmetic errors to visualize are those caused by carryless addition of an error vector. Indeed all arithmetic errors can be viewed as such although it is not an intuitively satisfying way of doing so. From the preceding discussions on distance it should be clear that all single faults will be detected by arithmetic codes as long as the circuit distance is small enough. Minimally redundant arithmetic codes have only an extra bit compared with parity codes and can be used easily in data transfers as well as performing arithmetic without internal changes to adders. Other types of non-arithmetic mappings are difficult and not well defined. Unordered codes can detect all unidirectional faults. This set includes single faults as well as a subset of arithmetic faults. These codes are generally too redundant and difficult to use under the single fault assumption. Arithmetic with unordered codes has been restricted essentially to two-rail codes. Fixed-weight codes have the

additional advantage of detecting all asymmetric errors. This is especially handy in memories where the nature of faults is such that asymmetric faults will always produce asymmetric errors.

The remainder of this chapter looks at the problem of translation of codes including decoders and encoders. Such an approach is justified in the following section.

5.2. On the Use of Different Fault Models

In this thesis we examined techniques to construct totally self-checking circuits using different codes and different fault models. Although this approach may seem quite incoherent, it is, in our opinion, justified by the different fault behaviours of different sections of a digital system. This is illustrated in another context, at the interface of computers and digital communication channels. Once the communication errors have been detected and corrected, all, or most, of the redundant check bits, are eliminated for further processing by the computer. Similarly it is believed that fault patterns in memories are quite different from those occurring in random logic, and they both bear little resemblance to those arising in peripherals. Unidirectional faults are often quoted as common to memory and busses. Arithmetic codes have been devised to check for errors that are intrinsic to adders. Parity codes will do well in both cases if single faults are assumed. This points to the interesting situation where the checker has a very different fault behaviour of that of the unit being checked.

Three approaches to the problem can be taken. In the first the designer selects a code which is good for a particular section and uses it throughout. The result will probably be hardship (sometimes intolerable) in designing the other sections and poor protection in some of them. In the

second situation, a compromise is made: the designer selects a code around which the whole machine can be designed fairly easily. The result is likely to be a mediocre code and/or mediocre error coverage in every section. Mediocre error coverage refers both to expensive waste or insufficient protection. In a third approach one can use the "best" codes in every section and translate them as the information is passed from one to another. This method does not compromise error protection in any section and eliminates potential waste present in the form of unnecessary redundancy. The trade-off is, of course, in the cost of designing, and implementing code translators.

5.3. General Considerations About Translators

A code translator is a functional block that maps codewords of a certain type onto codewords of a different type. Translators form a very large class of circuits but excludes specifically circuits with more than a single input code space. A checker can be viewed as a special case of code translator where code inputs are mapped onto code outputs of smaller size. A checker for m-out-of-n codes can be seen to perform a translation onto the smallest Berger code 01 and 10 (which also happens to be the 2-rail code). However this is just a special case. Another special situation arises in the case of encoders and decoders. At one point or another, especially in non-byte oriented machines, the information bits must be encoded, processed and eventually decoded. Encoders and decoders have input and output code spaces exactly equal to the input and output spaces, respectively. In a totally self-checking system all these functional blocks must satisfy certain properties to preserve the circuit's checking abilities. However there may be some problems especially in the case of encoders and decoders. This section examines some of those.

A code translator maps a code space onto another code space. From a structural point of view the translator is located at the boundary of functional blocks that have different fault models (typically). The question that comes up naturally is what is a valid fault model for the code translator. The answer is contigent mainly on two factors:

-The method used to translate the codes (i.e. structural considerations).

-The respective capabilities of the input and output codes.

For example if the check digit of a residue code is computed using adders then the appropriate fault model is expected to produce arithmetic errors. On the other hand if translation from parity codes to unordered codes is performed, the translator is not expected to be more than fault secure with respect to single faults unless some structural constraints are imposed (e.g. inverter-freeness).

It is recognized that the TSC properties do not exist beyond hardcore and non code spaces. For that reason we have to define TSC decoders as having outputs that are coded in a certain manner. The obvious way to proceed is to encode the outputs of decoders using systematic codes. After checking is performed, the check bits can be discarded and the information bits used subsequently for non TSC functions. Typically not much can be done about encoders as anything connected to their inputs is by definition unprotected. It is however possible to have a circuit structure that will guarantee a correct coded output (i.e. a fault-secure encoder) as well as being self-testing. For systematic codes one proceeds as follows: a checker connected to the inputs of the encoder and to the output check bits will determine whether the encoder has interpreted correctly the input lines (and not that the input lines were correct). For non-systematic codes the coded value must

first be decoded so it can be compared with the input lines. Again such a test confirms only that the encoder produces the right codeword of what it sees at its inputs.

It has just been seen that it is possible to imbed easily in TSC systems encoders and decoders if systematic codes are used throughout. Moreover a large class of translators can be related to the design of encoders:

Theorem 5.1: The TSC translation of systematic codes A into systematic code B is equivalent to the design of TSC encoders for code B.

Proof: Once code A is checked its check bits may be discarded and its information bits used to produce the check bits of code B as described before. □

Of course we do not have to discard the check bits. Indeed it is a good idea to use them if the mapping is from A to B and B is a subcode of A. Such a situation was presented before in the translation of Berger codes into parity codes. But it is the exception rather than the rule: the information content of the check bits is in general not easily transferable from one code to another.

Example 5.1: In the first chapter it was shown that fixed-weight codewords can be classified into cycling classes and suggested that it could be used to decode them serially. This example illustrates how it could be done in practice (see Figure 5.1). The cyclic codewords are loaded into a recycling register. Each of the flip-flops is connected to a collection of AND gates that implement the m-vertices of one congruence class. Only one of those will be high at any time thus generating a 1-out-of-(number of classes) code which can be checked by a TSC checker. This value along with the encoded number of shifts (i of T^{i}) performed is loaded into a block whose function is to extract the meaningful information. If its output is encoded then it is TSC. Note

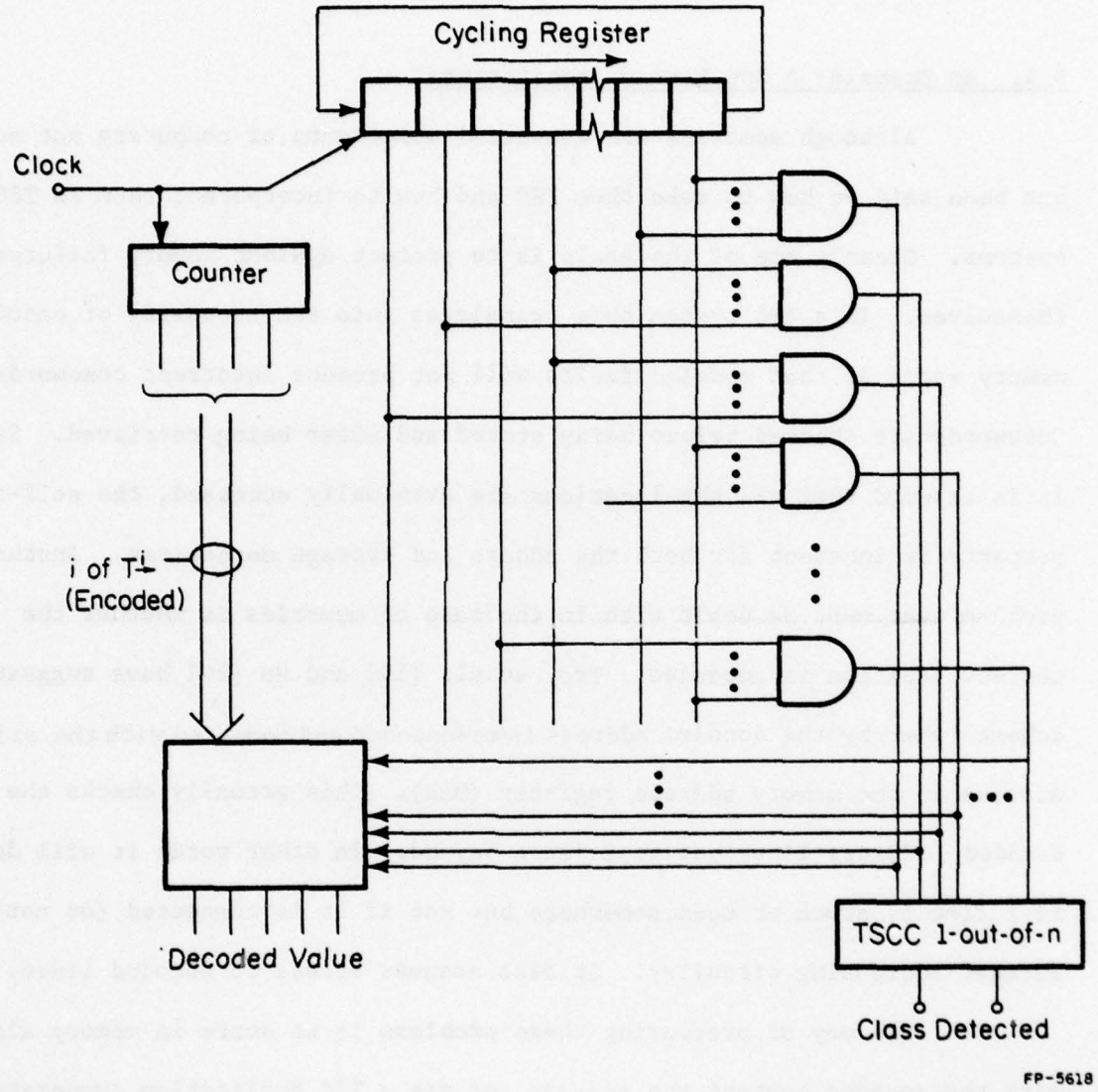


Figure 5.1. TSC sequential decoder for m-out-of-n codes.

that some tuning is required for the clock, counter (a sequential machine that can be made TSC using coding techniques [34]) and the control signals to be TSC.

5.4. An Example: A TSC Memory Architecture

Although memories are essential components of computers not much has been said on how to make them TSC and how to incorporate them in TSC systems. Clearly one of the goals is to protect against memory failures themselves. In a TSC system this translates into the necessity of encoding memory words so that modeled faults will not produce incorrect codewords. Codewords are checked before being stored and after being retrieved. Since it is assumed that all the locations are eventually accessed, the self-testing property is inherent for both the adders and storage mechanisms. Another problem that must be dealt with in the case of memories is whether the correct location is accessed. Troy et al. [15] and Ho [24] have suggested schemes whereby the decoded address is re-encoded and compared with the original address in the memory address register (MAR). This actually checks the decoded address lines but no failure beyond. In other words it will detect if a line is stuck or open somewhere but not if it is connected (or not) to further addressing circuitry. It also assumes access to decoded lines.

One way of overcoming these problems is to store in memory along with the encoded content the address and use a TSC duplication comparator to check if the desired and stored address match. In this situation re-encoding circuitry is exchanged for more memory. In line with this chapter i.e. considering different fault models, we shall discuss other and more appealing alternatives. In TSC systems the address that eventually gets to the memory address register is encoded. This redundancy can effectively

be used to check whether or not the content gated into the memory buffer register (MBR) is from the correct location in the following manner: Instead of storing a complete copy of the address one can store only its check bits. If the code is non-systematic and/or not redundant enough to guard against failure modes in the memory (failure modes cannot be easily altered to fit the codes) then an appropriate code translator is required. Appropriate refers, according to the case, to a translator that will map the non-systematic code into a systematic one and/or a systematic code into another systematic code with a sufficient amount of redundancy. For example the address lines may be encoded using parity check. If memory failures are assumed to be unidirectional, the scheme presented here will call for generating check bits for the Berger codeword of this address and store it in memory along with the encoded data. This architecture, that assumes unidirectional faults in memory and single fault elsewhere, is described in Figure 5.2.

It is still possible to object to the amount of redundancy used as memories tend to be expensive. If it is assumed that single fault may occur in either the addressing circuitry or in the memory content but not both, or in the case of unidirectional faults, that the faults will be in the same direction in both the memory and addressing circuits, then it is possible to protect both address and data with very little redundancy more than what is needed to check data only. In the single fault case this approach is suggested: store in memory whatever data is desired and attach to it the sum mod 2 of the parity of this data and of the parity of the address. To protect from unidirectional faults in both address and data, one can store the Berger code check bits of the concatenation of the address and data along with the data itself in memory. The amount of redundancy required in this case may be slightly larger than the redundancy to protect solely the content. This is

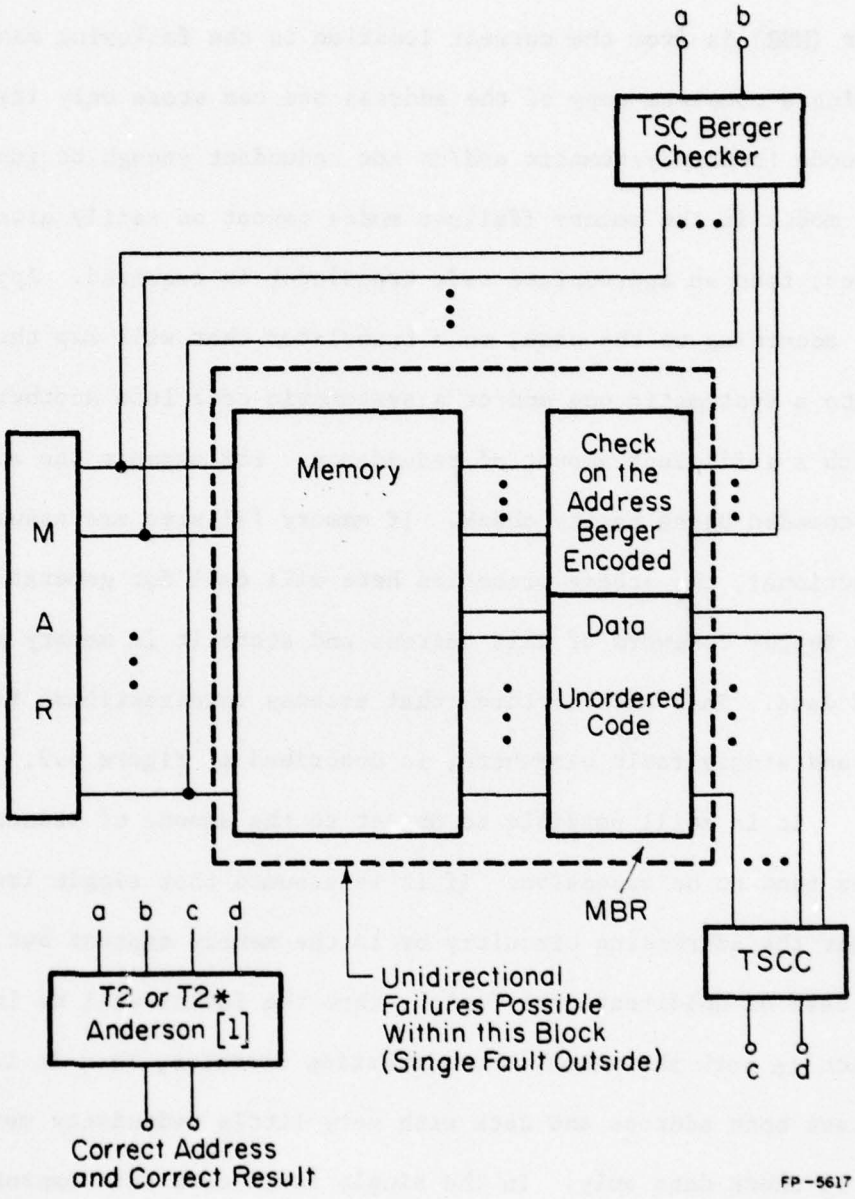


Figure 5.2. A TSC memory.

because the number of check bits required increases by one each time the amount of information bits doubles. If both parity and Berger codes are used then, using similar ideas, one has to store the Berger check bits with its least significant bit modified to reflect the parity of the other code-word. That is simply the sum mod 2 of the parity bit (of the parity encoded word (address and data)) and the least significant bit of the Berger encoded word. The resulting parity bit will detect all single faults in the word formed by the concatenation of the address and data and the remaining check bits will detect unidirectional faults in the appropriate word. In this circumstance the redundancy of the concatenated word is the same as in the Berger encoded word alone.

Finally it can be said that the process of writing into and reading out of memory is code disjoint so checking may be done only after reading so as to detect writing errors as well. However many memory architectures show a dual purpose memory buffer register that stores data to be written into or read out of memory; therefore checking is achieved before writing and after reading at no extra cost.

5.5. The Problem of Functional Mappings

In general, codes are devised so they suit a particular structure that realizes a well defined function. It turns out that codes are essentially special purpose and very cumbersome for situations other than that they were designed for. The problem of functional mappings of code spaces can be equated to the problem of translation followed by functional mapping of non-code or systematically coded spaces as seen in the following steps:

As before the A_i's represent code space inputs which are either equal or translated into a systematic code space with information I_i and check C_i.

$$A_1 \times A_2 \times \ldots \times A_k \overset{=}{\rightarrow} (I_1,C_1) \times (I_2,C_2) \times \ldots \times (I_k,C_k)$$

$$\rightarrow (I_1 \times I_2 \times \ldots \times I_k, C_1 \times C_2 \times \ldots \times C_k) \qquad (*)$$

The functional mapping is performed next. Let O_i be the information at the output with checks ζ_i and final coded output B_i.

$$(*) \rightarrow (O_1 \times O_2 \times \ldots \times O_j, \zeta_1 \times \zeta_2 \times \ldots \times \zeta_j)$$

$$\rightarrow (O_1,\zeta_1) \times (O_2,\zeta_2) \times \ldots \times (O_k,\zeta_j)$$

$$\overset{=}{\rightarrow} B_1 \times B_2 \times \ldots \times B_j.$$

If the C_i are non-existent then the process is not self-checking. We can now formalize the idea of codes and the structure that fits them best. This occurs when

given $\quad I_1 \times I_2 \times \ldots \times I_k \overset{\phi}{\rightarrow} O_1 \times O_2 \times \ldots \times O_j$

then $\quad C_1 \times C_2 \times \ldots \times C_k \overset{\phi}{\rightarrow} \zeta_1 \times \zeta_2 \times \ldots \times \zeta_j$ for systematic codes

or $\quad A_1 \times A_2 \times \ldots \times A_k \overset{\phi}{\rightarrow} B_1 \times B_2 \times \ldots \times B_j$ for non-systematic codes.

For data transfer ϕ is simple transmittance (works well for all codes); for arithmetic codes ϕ can be any operation in modular arithmetic. Unordered codes are such that the redundancy essentially contains information about the weight of the codeword. Logical and arithmetic operations in general are not weight preserving. Therefore it is extremely difficult to map unordered codes in general. The intuitive ways of doing it have more in common with abstract automata theory than they have with practical switching theory. Except for the pathological case of the 2-out-of-4 codes which can be mapped one-to-one onto the integers 0 to 5 using weight 3,2,1,-1, it is felt that no such mappings exist in general.

Therefore the problem of processing encoded information is essentially the same as decoding and then processing. The 2-step approach may be conceptually simpler however it is typically not easier, as seen in Section 5.3.

5.6. Concluding Remarks

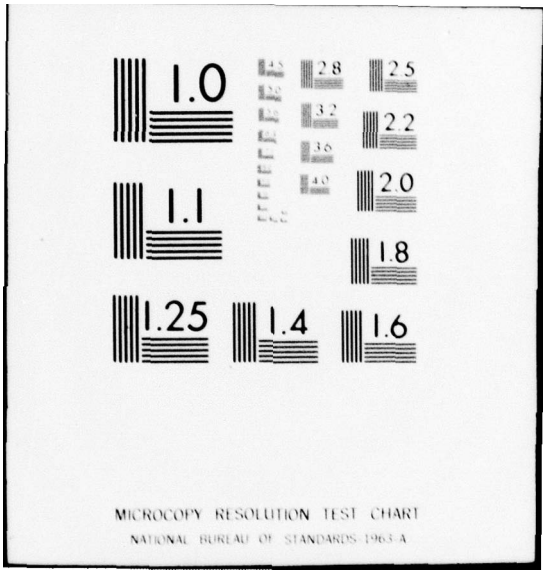
Code translation has already been studied formally for Smith [52]. In this section it has been suggested that the use of different codes for different blocks is advantageous in TSC systems, hence the need for code translators. Moreover translation for systematic codes is essentially equivalent to encoding and decoding. For most systematic codes methods to systematically obtain the check bits are known and very similar to the techniques used for checking. For example, the residue bits can be obtained from modulo adders and the parity bit, from a tree of exclusive-OR gates. These simple properties underscore the desirability of systematic codes in general. Non-systematic code translation is generally more efficient. In the case of AN code the codewords can be obtained from a series of ADD and SHIFT operations; decoding is more expensive as division requires use of the ALU. More difficult is the manipulation of fixed-weight codes.

AD-A056 280 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
ON THE DESIGN OF SELF-CHECKING SYSTEMS UNDER VARIOUS FAULT MODE--ETC(U)
OCT 77 J DUSSAULT DAAB07-72-C-0259

UNCLASSIFIED R-791 NL

2 OF 2
AD
A056 280

END
DATE FILMED
8-78
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

6. CONCLUSION

6.1. Totally Self-Checking with Respect to the Pin Fault Model

The pin fault model was introduced by Ketelsen [27] and assumes that stuck-at faults occur only at the input or output pins of integrated circuit modules. The pin fault model includes multiple as well as single stuck-at pin faults. The pin fault model is justified by a variety of reasons. Available failure data for digital IC's indicates that lead bond failures are predominant. These result from either packaging defects or from stresses suffered during normal operation. These stresses are either mechanical, environmental or electrical. It is important to separate failures according to the period of time in which they occur. If most pin faults occur during the infancy of a product then the pin fault model is useful mainly for devising initial test sets and has very limited value in a self-checking system. If most faults occur during the normal life of a digital system then it may be interesting to consider the design of TSC systems with respect to a pin fault set.

However, there are some problems. As one may expect, the multiple fault assumption complicates the picture. It is known that a test set for a module realizing $G(\chi,\phi)$ exists if for every input x_i there exists test vectors t_i and t_j which differ only in the value applied to x_i and have $G(t_j,\phi) \neq G(t_i,\phi)$. This test set exists for all non-vacuous functions G. Application of pairs of vectors is checked by comparing with the expected value. Ideally one would like to sequence the tests so that the output alternate; failure to do so indicates an error. As far as designing self-checking circuits it implies (since vectors are applied randomly)

a) That there must be some way of recognizing that a valid pair of test vectors is being applied to the circuit.

b) That a correct pair of coded ouput vectors is also being generated.

Since multiple faults are assumed there does not exist a way from within the chip to determine that indeed such a pair of vectors is being applied. Duplication of the basic modules plus logic on each to detect a valid test pair seems to be the only way to go. If it is indeed the case the fault set may as well be enlarged considerably so as to justify the cost of duplication. Therefore, the pin fault model does not seem to be a reasonable fault model from the point of view of TSC design.

6.2. Summary of Thesis

In this thesis we have looked at a variety of codes and their associated circuit structures. Parity, unordered and arithmetic codes were the main ones discussed. Each one fit a particular fault model very well but interchangeability was shown to be difficult. At the system level the use of translators was suggested.

Chapter 2 was an investigation of the algebraic structure of unordered codes. Unordered codes were shown to have common construction rules whether separable or not. In addition, fixed-weight codes exhibited nice class properties, especially those with higher rates.

Some of these results were carried into Chapter 3 where they were used to provide some ideas about the partitioning problems related to the design of checkers. Some results were established concerning the design of checkers for Berger codes. Finally some practical considerations were given to checking under the single fault assumption and also for asymmetric errors.

Chapter 4 discussed the problems encountered when designing arithmetic circuits using arithmetic codes. As long as certain rules are obeyed conventional checkers for arithmetic codes can be used but with a substantial increase in hardcore. Ideal checkers were exemplified by a residue 3 and 3N checker but generalized results have not been obtained.

The main topic of Chapter 5 was code translation. It was suggested that in a system with several different blocks and fault models, the best way of taking advantage of coding techniques is to use the best codes for each of the sections and translate them as information is passed from a section to another. Separable codes were preferred for translation purposes but may not be as good as non-systematic codes.

6.3. Topics for Further Research

In this thesis we have tried to solve some problems and have obtained a mixture of results. Some are perhaps promising, some are clearly negative. Perhaps the result that eluded this research most was methods to map codes that were meant for some specific applications and used in different circumstances. The whole study of Chapter 2 was originally aimed at that problem. Hence the idea of universal codes. But we know that there are no such thing as codes for logical operations. One area of interest is along the lines suggested by Carter et al. [14] where they use morphic Boolean functions: simply encode each input and replace gates by their morphism that will preserve information and protection. The problem is especially difficult since we are competing with duplication schemes.

Berger codes have the desired property of being separable. However their checkers do not have the simplicity nor the structure of

combinatorial checkers. Good checkers would be handy especially since those codes feel better to use.

Arithmetic codes have been shown to perform reasonably well in a TSC environment. Perhaps there exist less conventional structures that could perform checking and reduce the size of the hardcore to a minimum.

This thesis has discussed coding techniques to detect errors. It is possible to introduce more redundancy and be able to correct some errors. By placing checkers appropriately it may be possible to keep the system self checking. Such systems may be competitive with various forms of multiple redundancy. It is also felt that systems with interactive recovery should also be looked at from that point of view e.g. duplicated systems with encodings.

REFERENCES

1. Anderson, D. A., "Design of Self-Checking Digital Networks Using Coding Techniques," Coordinated Science Laboratory Report R-527, University of Illinois, September 1971.

2. Anderson, D. A. and Metze, G., "Design of Totally Self-Checking Check Circuits for m-out-of-n Codes," IEEE Transactions on Computers, Vol. C-22, pp. 263-265, March 1973.

3. Ashjaee,M. and Reddy, S. M., "Totally Self-Checking Checkers for a Class of Separable Codes," Proc. of the 12th Annual Allerton Conference on Circuit and System Theory, Monticello, Illinois, pp. 238-243, October 1974.

4. Avizienis, A., "Arithmetic Algorithms for Error-Coded Operands," Digest 1972 International Symposium on Fault-Tolerant Computing, Pasadena, December 1971.

5. Avizienis, A., et al., "The STAR Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," IEEE Transactions on Computers, Vol. C-20, pp. 1312-1321, November 1971.

6. Avizienis, A., "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design," IEEE Transactions on Computers, Vol. C-20, pp. 1322-1331, November 1971.

7. Bark, A. and Kinne, C. B., "The Application of Pulse Position Modulation to Digital Computers," Proc. National Electronics Conf., pp. 656-664, September 1953.

8. Berger, J. M., "A Note on Error Detection Codes for Asymmetric Channels," Information and Control, Vol. 4, pp. 68-73, March 1961.

9. Betancourt, R., "Derivation of Minimum Test Sets for Unate Logical Circuits," IEEE Transactions on Computers, Vol. C-20, pp. 1264-1269, November 1971.

10. Birkhoff, G., "Lattice Theory," 3rd Edition, Department of Mathematics, Harvard University, Cambridge, 1963.

11. Carter, W. C. and Schneider, P. R., "Design of Dynamically Checked Computers," IFIP 68, Vol. 2, Edinburgh,Scotland, pp. 878-883, August 1968.

12. Carter, W. C., Bouricius, W. G., Jessep, D. C., Roth, J. P., Schneider, P. R., and Wadia, A. B., "A Theory of Design of Fault-Tolerant Computers Using Standby Sparing," Digest 1971 Symposium on Fault-Tolerant Computing, Pasadena, pp. 83-86, March 1971.

13. Carter, W. C., et al., "Logic Design for Dynamic and Interactive Recovery," IEEE Transactions on Computers, Vol. C-20, pp. 1300-1311, November 1971.

14. Carter, W. C., Wadia, A. B., and Jessep, D. C., "Implementation of Checkable Acyclic Automata by Morphic Boolean Functions," Proc. of the Symposium on Computer and Automata, Polytechnic Institute of Brooklyn, April 13-15, 1971.

15. Cook, R. W., et al., "Design of a Self-Checking Microprogram Control," IEEE Transactions on Computers, Vol. C-22, No. 3., pp. 255-262, March 1973.

16. Dussault, J. and Metze, G., "A Low-Cost Totally Self-Checking Checker for 3N and Residue 3 Codes," Proc. of the 14th Annual Allerton Conference on Circuit and System Theory, Monticello, September 30-October 1, 1976.

17. Diaz, M., "Design of Totally Self-Checking and Fail-Safe Sequential Machines," The Fourth Annual International Symposium of Fault-Tolerant Computing, pp. 3-19 to 3-24, June 1974.

18. Elias, P., "Computation in the Presence of Noise," IBM Journal of Research and Development, Vol. 2, pp. 346-353, October 1958.

19. Freiman, C. V., "Protective Block Codes for Asymmetric Binary Channels," Columbia University, Ph.D. Thesis, May 1961.

20. Freiman, C. V., "Upper Bounds for Fixed Weight Codes of Specified Minimum Distance," IEEE Transactions on Information Theory, pp. 246-248, July 1964.

21. Garner, H. L., "Error Codes for Arithmetic Operations," IEEE Transactions on Electronic Computers, Vol. EC-15, pp. 763-770, October 1966.

22. Hall, M., "A Survey of Combinatorial Analysis," Some Aspects of Analysis and Probability, Vol. IV, John Wiley, 1958, pp. 77-104.

23. Ho, D. S., "The Study of a Totally Self-Checking Adder," Coordinated Science Laboratory Report R-582, University of Illinois, August 1972.

24. Ho, D. S., "The Design of Totally Self-Checking Systems," Coordinated Science Laboratory Report R-723, University of Illinois, April 1976.

25. Kautz, W. H. and Elspas, B., "Single-Error-Correcting Codes for Constant Weight Data Words," IEEE Transactions on Information Theory, Vol. IT-11, pp. 132-141, January 1965.

26. Kohavi, I., "Fault Diagnosis in Logical Circuits," Conf. Rec. of the 1969 Tenth Annual Symposium on Switching and Automata Theory, pp. 166-173, October 1969.

27. Ketelsen, M. L., "An Integrated Circuit Fault Model for Digital Systems," Coordinated Science Laboratory Report R-743, September 1976.

28. Kolupaev, S. G., "Self-Testing Residue Trees," Technical Report No. 49, Digital Systems Laboratory, Stanford University, August 1973.

29. Kuhl, J. G. and Reddy, S. M., "Design of Asynchronous Circuits - Some Problems," Proceedings of the 14th Allerton Conference, Monticello, pp. 191-200, 1976.

30. Massey, J. L., "Survey of Residue Coding for Arithmetic Errors," ICC Bulletin, Vol. 3, Rome, Italy, pp. 195-203, October 1964.

31. Massey, J. L. and Garcia, O. M., "Error-Correcting Codes in Computer Arithmetic," Advances in Information Sciences, Chap. 5.

32. Metze, G. and Smith, J. E., "The Design of Totally Self-Checked Combinational Logic Circuits," Proc. of the 1976 Conf. on Information Sciences and Systems, Baltimore, Maryland, 1976.

33. Monteiro, P. and Rao, T.R.N., "A Residue Checker for Arithmetic and Logical Operations," Digest of the 1972 Symposium on Fault-Tolerant Computing, Newton, MA., pp. 8-13, June 1972.

34. Özgüner, F., "Design of Totally Self-Checking Asynchronous Sequential Machines," Coordinated Science Laboratory Report R-679, May 1975.

35. Paige, M. R., "Generation of Diagnostic Tests Using Prime Implicants," Coordinated Science Laboratory Report R-414, University of Illinois, May 1969.

36. Peterson, W. W., "On Checking an Adder," IBM Journal of Research and Development, Vol. 2, pp. 166-168, April 1958.

37. Peterson, W. W. and Rabin, M. O., "On Codes for Checking Logical Operations," IBM Journal of Research and Development, Vol. 3, pp. 163-168, April 1959.

38. Peterson, W. W., Error-Correcting Codes, The M.I.T. Press, Cambridge, 1972.

39. Pierce, W. H., Failure Tolerant Computer Design, Academic Press, New York, 1965.

40. Pitt, D. A., "Design of Totally Self-Checking Asynchronous Sequential Machines," Report No. UIUCDCS-R-73-593, University of Illinois, September 1973.

41. Pradhan, D. K. and Reddy, S. M., "A Design Technique for the Synthesis of Fault-Tolerant Adders,"Digest of the 1972 Symposium on Fault-Tolerant Computing, Newton, MA., pp. 20-24, June 1972.

42. Rao, T.R.N., "Biresidue Error-Correcting Codes for Computer Arithmetic," IEEE Transactions on Computers, Vol. C-15, pp. 398-402, May 1970.

43. Rao, T.R.N. and Garcia, O. N., "Cyclic and Multiresidue Codes for Arithmetic Operations," IEEE Transactions on Information Theory, Vol. IT-17, pp. 85-91, January 1971.

44. Reddy, S. M. and Wilson, J. R., "Easily Testable Cellular Realizations for the (Exactly P)-out-of-n and (P or More)-out-of-n Logic Functions," IEEE Transactions on Computers, Vol. C-23, pp. 98-100, January 1974.

45. Reddy, S. M., "A Note on Self-Checking Checkers," IEEE Transactions on Computers, Vol. C-23, pp. 1100-1102, October 1974.

46. Reddy, S. M. and Ashjaee, M. J., "On Totally Self-Checking Checkers for Separable Codes," Proc. of the 1976 International Symposium on Fault Tolerant Computing, pp. 151-156, June 21-23, 1976.

47. Reynolds, D. A., "The Design of Alternating Logic Systems with Fault Detection Capabilities," Coordinated Science Laboratory Report R-738, August 1976.

48. Sellers, F. F., Hsiao, M. Y. and Bearnson, L. W., Error Detecting Logic for Digital Computers, McGraw-Hill, 1968.

49. Shannon, C. E. and Moore, E. F., "Reliable Circuits Using Less Reliable Relays," J. of the Franklin Institute, Vol. 262, 191-208 (Sept. 1956), 281-297 (Oct. 1956).

50. Smith, J. E. and Metze, G., "General Design Rules for the Construction of m-out-of-n Totally Self-Checking Checkers," Coordinated Science Laboratory Report R-693, University of Illinois, October 1975.

51. Smith, J. E. and Metze, G., "General Design Rules for the Construction of m-out-of-n Totally Self-Checking Checkers," Proc. of the 13th Annual Allerton Conference on Circuit and System Theory, Monticello, Illinois, pp. 704-715, October 1975.

52. Smith, J. E., "The Design of Totally Self-Checking Combinational Circuits," Coordinated Science Laboratory Report R-737, University of Illinois, August 1976.

53. Von Neumann, J., "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," Annals of Mathematical Studies, No. 34, pp. 43-98, Princeton University Press, Princeton, N.J., 1956.

54. Winograd, S., "Coding for Logical Operation," IBM Journal of Research and Development, Vol. 6, pp. 430-436, October 1962.

55. Wakerly, J. F., "Checked Binary Addition Using Parity Prediction and Checksum Codes," Technical Note No. 39, Digital Systems Laboratory, Stanford University, January 1974.

56. Wakerly, J. F., "Partially Self-Checking Circuits and Their Use in Performing Logical Operations," IEEE Transactions on Computers, Vol. C-23, pp. 658-666, July 1974.

57. Wakerly, J. F. and McCluskey, E. J., "Design of Low-Cost General Purpose Self-Diagnosing Computers," Technical Note No. 38, Digital Systems Laboratory, Stanford University, January 1974.

58. Woodard, S. and Metze, G., "Self-Checking Alternating Logic: Combinational Network Analysis," Proc. of the 15th Allerton Conference, Monticello, Illinois, September 28-30, 1977.

VITA

Jean Dussault was born in Hull, Québec on June 4, 1953. He received his B.A. Sc. (1973) and M.A. Sc. (1974) degrees in electrical engineering both from the University of Ottawa. From 1973 to 1977 his work was supported by graduate scholarships from the Province of Québec. From 1974 to 1977 he was a research fellow with the Digital Systems Group at the Coordinated Science Laboratory.